*Join the discussion @ p2p.wrox.com*




Wrox Programmer to Programmer™

.NET开发经典名著



Professional C# 7 and .NET Core 2.0

# C#高级编程（第11版）

## C# 7 & .NET Core 2.0

[美] 克里斯琴•内格尔(Christian Nagel)　著

李　铭　译

清华大学出版社

.NET 开发经典名著

# C#高级编程

## (第 11 版)

## C# 7 & .NET Core 2.0

[美] 克里斯琴 • 内格尔(Christian Nagel)　著
李　铭　译

清华大学出版社
北　京

Christian Nagel
Professional C# 7 and .NET Core 2.0
EISBN: 978-1-119-44927-0


图书在版编目(CIP)数据

C#高级编程：第11版：C# 7 & .NET Core 2.0 / (美)克里斯琴 • 内格尔(Christian Nagel) 著；李铭 译. —北京：清华大学出版社，2019
(.NET 开发经典名著)
书名原文：Professional C# 7 and .NET Core 2.0
ISBN 978-7-302-52256-0

Ⅰ. ①C…　Ⅱ. ①克… ②李…　Ⅲ. ①C 语言—程序设计　Ⅳ. ①TP312.8

中国版本图书馆 CIP 数据核字(2019)第 016880 号

**责任编辑：**王　军　于　平
**装帧设计：**孔祥峰
**责任校对：**成凤进
**责任印制：**董　瑾

**出版发行：**清华大学出版社
**网　　址：**http://www.tup.com.cn，http://www.wqbook.com
**地　　址：**北京清华大学学研大厦 A 座　　**邮　　编：**100084
**社 总 机：**010-62770175　　**邮　　购：**010-62786544
**投稿与读者服务：**010-62776969，c-service@tup.tsinghua.edu.cn
**质 量 反 馈：**010-62772015，zhiliang@tup.tsinghua.edu.cn
**印 装 者：**三河市龙大印装有限公司
**经　　销：**全国新华书店
**开　　本：**190mm×260mm　　**印　　张：**67.25　　**字　　数：**2273 千字
**版　　次：**2019 年 3 月第 1 版　　**印　　次：**2019 年 3 月第 1 次印刷
**定　　价：**198.00 元

---

产品编号：079458-01

# 译　者　序

C#是微软公司在 2000 年 6 月发布的一种面向对象的、运行于.NET Framework 之上的高级程序设计语言，由 Anders Hejlsberg 主持开发，它是第一个面向组件的编程语言，其源码会编译成 MSIL 再运行。C#是微软公司 .NET Framework 的主角。C#是一种安全的、稳定的、简单的、优雅的、由 C 和 C++衍生出来的编程语言。它在继承 C 和 C++强大功能的同时，去掉了它们的一些复杂特性，使程序员可以快速地编写各种基于微软.NET 平台的应用程序。

C#是兼顾系统开发和应用开发的最佳实用语言，很有可能成为编程语言历史上的第一个“全能”型语言。它提供了以下软件工程要素的支持：强类型检查、数组维度检查、未初始化的变量引用检测、自动垃圾收集。目前，C#已经发布到 7.2 版本，其中：

- C# 1.0：纯粹的面向对象。
- C# 2.0：增加了泛型编程新概念。
- C# 3.0：率先实现了 LINQ 的语言。
- C# 4.0：支持动态编程。
- C# 5：支持异步编程。
- C# 6 和.NET Core 1.0 提供了代码的共享，.NET Core 运行在 Windows、Linux 和 Mac 操作系统上。因此自从.NET Core 推出以来，就可以在任何操作系统上构建程序。
- C# 7 和.NET Core 2.0 提供了函数式编程。

随着微软加快了发布速度，同时在每次更新中都提供了更重要的改进，对新工具和新特性的快速处理就变得前所未有的重要。《C#高级编程(第 11 版) C#7&.NET Core 2.0》就是为了达到这个目的而设计的，关于 C#的一切都在这里。

本书为有经验的程序员提供了他们需要与世界领先的编程语言有效合作的信息。最新的 C 语言增加了许多新的特性并更新了一些特性，帮助你在更短的时间内完成更多的工作，本书是快速入门的理想指南。C# 7 重点关注代码简化和性能，对本地函数、元组类型、记录类型、模式匹配、非可空引用类型、不可变类型提供了新的支持。Visual Studio 的改进将给 C 开发人员与空间交互的方式带来重大改变，将.NET 引入非微软平台，并将 Docker、GULP 和 NPM 等工具与其他平台结合起来。

本书分四个部分，介绍 C#语言及其在各个领域中的应用。第 I 部分给出 C#语言的良好背景知识。第 II 部分介绍独立于应用程序类型的.NET Core 和 Windows Runtime。第 III 部分论述 Web 应用程序和服务。第 IV 部分介绍如何使用 XAML 构建应用程序，包括 Universal Windows 应用程序和 Xamarin 应用程序。

无论你是 C#新手，还是刚刚迁移到 C# 7，如果希望对最新特性有一个扎实的掌握，能够利用该语言的全部功能来创建健壮的、高质量的应用程序，本书都是你需要知道的所有内容的一站式指南。

在这里要感谢清华大学出版社的编辑，他们为本书的翻译投入了巨大的热情，并付出了很多心血。没有你们的帮助和鼓励，本书不可能顺利付梓。本书主要章节由李铭翻译，参与翻译的还有陈妍、何美英、陈宏波、熊晓磊、管兆昶、潘洪荣、曹汉鸣、高娟妮、王燕、谢李君、李珍珍、王璐、王华健、柳松洋、曹晓松、陈彬、洪妍、刘芸、邱培强、高维杰、张素英、颜灵佳、方峻、顾永湘、孔祥亮。

对于这本经典之作，译者本着“诚惶诚恐”的态度，在翻译过程中力求“信、达、雅”，但是鉴于译者水平有限，错误和失误在所难免，如有任何意见和建议，请不吝指正。

译者

# 作 者 简 介



Christian Nagel 是 Visual Studio 和开发技术方向的 Microsoft MVP，担任微软开发技术代言人(Microsoft Regional Director)已经超过 15 年。Christian 是 CN innovation 公司的创始人，CN innovation 公司提供指导、培训、代码评审，并协助使用微软技术设计和开发解决方案。他拥有超过 25 年的软件开发经验。

Christian Nagel 最初在 Digital Equipment 公司通过 PDP 11 和 VAX / VMS 系统开始他的计算机职业生涯，接触过各种语言和平台。在 2000 年，.NET 只有一个技术概览版时，他就开始使用各种技术建立.NET 解决方案。目前，他主要指导人们开发和设计 Windows 应用程序、ASP.NET Core Web 应用程序和 Xamarin，并帮助他们使用 Microsoft Azure 服务产品。

在软件开发领域工作多年以后，Christian 仍然热爱学习和使用新技术，并通过多种形式教别人如何使用新技术。他的 Microsoft 技术知识非常渊博，编写了很多书，拥有微软认证培训师(MCT)和微软认证解决方案开发专家(MCSD)认证。Christian 经常在国际会议(如 Microsoft Ignite、BASTA！和 TechDays)上发言。他创立了 INETA Europe 来支持.NET 用户组。他的联系网站是 www.cninnovation.com，博客是 https://csharp.christiannagel.com，其 Twitter 账号是@christiannagel。

# 技术编辑简介



**István Novák** 是 SoftwArt 的合伙人和首席技术顾问，SoftwArt 是匈牙利的一家小型 IT 咨询公司。István 是一名软件架构师和社区的传教士。在过去 25 年里，他参加了五十多个企业软件开发项目。2002 年，他在匈牙利与他人合作出版了第一本关于.NET 开发的图书。2007 年，他获得微软最有价值专家(MVP)头衔。2011 年，他成为微软开发技术代言人(Microsoft Regional Director)。István 与他人合作出版了 *Visual Studio 2010 and .NET 4 Six-in-One*(Wiley, 2010)和《Windows 8 应用开发入门经典》(Wiley, 2012)，独立撰写了《Visual Studio 2010 LightSwitch 开发入门经典》(Wiley, 2011)。

István 从匈牙利的布达佩斯技术大学获得硕士学位和软件技术的博士学位。他与妻子和两个女儿居住在匈牙利的 Dunakeszi。他是一个充满激情的潜水员，在一年的任何季节都会去红海潜水。

# 致　　谢

我要感谢 Charlotte Kughen，他使本书的文本更具可读性。.NET Core 在不断演变，Charlotte 为我提供了巨大的帮助，使本书阐述的.NET 技术能与时俱进。她还利用许多周末的时间帮助本书快速出版。也特别感谢 István Novák，他作为技术编辑，校正了本书中的一些错误。

.NET Core 的飞速发展和我在书中使用的临时构建存在一些问题，István 向我挑战，改进了代码示例，让读者更容易理解。谢谢你们：Charlotte 和 István，你们让本书的质量上了一个大台阶。

我还要感谢.NET Core 团队的 Richard Lander。我们就 C#高级编程第 11 版的内容和方向进行了热烈的讨论。Rich 还抽出时间给我提供了一些关于本书中几个章节的好建议。

我也要感谢 Kenyon Brown，以及 Wiley 出版社帮助出版本书的其他人。我还要感谢我的妻子和孩子，为了撰写本书，我花费了大量的时间，包括晚上、周末和冬季假日，但你们很理解并支持我。Angela、Stephanie、Matthias 和 Katharina，你们是我深爱的人。没有你们，本书不可能顺利出版。

# 前　　言

许多年过去了，.NET 有了新的发展势头。.NET Framework 有一个年轻的兄弟.NET Core！

.NET Framework 是封闭的源代码，只能在 Windows 系统上使用。现在.NET Core 是开源的，可以在 Linux 上使用，并且使用现代模式。在.NET 生态系统中有很多巨大的改进。

**注意：**

由于最近的变化，C#在最受欢迎的编程语言中排名前十，而.NET Core 在最受欢迎的框架中排名第三。在 Web 和桌面开发人员中，C#在最流行的语言中排名第三。详情请登录 https://insights.stackoverflow.com/ survey/ 2017。

使用 C#和 ASP .NET Core，可以创建运行在 Windows、Linux 和 Mac 上的 Web 应用程序和服务。使用 Windows Runtime(Windows 运行库)，可以通过 C#和 XAML 以及.NET Core 创建本机 Windows 应用程序(也称为通用 Windows 平台，UWP)。通过 Xamarin，使用 C#和 XAML 可以创建运行在 Android 和 iOS 设备上的应用程序。在.NET Standard 的帮助下，可以创建能在 ASP.NET Core、Windows 应用、Xamarin 中共享的库，还可以创建传统的 Windows Forms 和 WPF 应用程序。所有这些都在本书中介绍。

本书大部分示例都建立在带有 Visual Studio 的 Windows 系统上。许多示例也在 Linux 上进行了测试，并在 Linux 和 Mac 上运行。除了 Windows 应用程序示例之外，还可以使用 Mac 的 Visual Studio Code 或 Visual Studio for the Mac 作为开发环境。

## 0.1　.NET Core 的世界

.NET 有很长的历史，但是.NET Core 很年轻。.NET Core 2.0 从.NET Framework 中获得了许多新的 API，使其更容易将现有的.NET Framework 应用程序迁移到.NET Core 的新世界。

一个简单的步骤是，可以创建使用.NET Standard 2.0 的库，这些库可以在.NET Framework 4.6.1 及以上版本的应用程序、.NET Core 2.0 应用程序和 Windows Build 16299 以上版本的应用程序中使用。

现在，没有理由不在后端使用 ASP.NET Core。随着迁移到.NET Standard 的简化，越来越多的库可以在.NET Core 中使用。总体来看，ASP.NET Core MVC 与 ASP.NET MVC 非常相似。然而 ASP.NET Core MVC 要灵活得多，使用.NET Core 模式时更容易操作，也更容易扩展。

对于创建新的 Web 应用程序，可能只需要使用新技术 Razor Pages。如果应用程序增长，Razor Pages 可以很容易地扩展到使用 ASP .NET Core MVC 的模型-视图-控制器模式。

在撰写本文时，用于实时通信的技术 SignalR 的.NET Core 版本即将发布。

ASP.NET Core 与 JavaScript 技术(如 Angular 和 React/Redux)的结合非常有效，甚至还有模板使用这些技术以及用于后端服务的 ASP.NET Core 创建项目。

**注意：**

可以通过 https://github.com/dotnet/corefx 访问.NET Core 的源代码。.NET Core 命令行可以在 https://github.com/dotnet/cli 上使用。在 https://github.com/aspnet 上有许多 ASP.NET Core 的存储库。其中包括 ASP.NET Core MVC、Razor、SignalR、EntityFrameworkCore 等。

下面是对.NET Core 部分特性的总结：

- .NET Core 是开源的。
- .NET Core 使用现代模式。
- .NET Core 支持在多个平台上开发。
- ASP.NET Core 可以在 Windows 和 Linux 上运行。

使用.NET Core 时，会发现这项技术是.NET 自从第一个版本以来最大的改变。.NET Core 是一个新的开始，从这里可以继续我们的旅程，快速进行新的开发。

## 0.2 C#的重要性

C#在 2002 年发布时，是一个用于.NET Framework 的开发语言。C#的设计思想来自 C++、Java 和 Pascal。Anders Hejlsberg 从 Borland 来到微软公司，带来了开发 Delphi 语言的经验。Hejlsberg 在微软公司开发了 Java 的 Microsoft 版本 J++，之后创建了 C#。

**注意：**

Anders Hejlsberg 现在已经转移到 *TypeScript*(而他仍在影响 C#)， Mads Torgersen 是 C#的项目负责人。C#的改进可以在 https://github.com/dotnet/csharplang 上公开讨论。在这里，可以阅读 C#语言建议和会议记录，也可以提交自己的 C#建议。

C#一开始不仅作为一种面向对象的通用编程语言，而且是一种基于组件的编程语言，支持属性、事件、特性(注解)和构建程序集(包括元数据的二进制文件)。

随着时间的推移，C#增强了泛型、语言集成查询(Language Integrated Query，LINQ)、lambda 表达式、动态特性和更简单的异步编程。C#编程语言并不简单，它提供了很多功能，而且实际使用的功能在不断进化。因此，C#不仅是面向对象或基于组件的语言，它还包括函数式编程的理念，开发各种应用程序的通用语言会实际应用这些理念。

在 C# 6 中，编译器的源代码完全重写了。不仅新的编译器管道可以在自定义程序中使用，微软还获得了一些新的资源，使变更不会破坏程序的其他部分。因此，增强编译器变得容易了。

C# 7 再次添加了许多具有函数编程背景的新特性，如本地函数、元组和模式匹配。

## 0.3 C# 7 的新特性

C# 6 扩展包括 static using、基于表达式体的方法和属性、自动实现的属性初始化器、只读自动属性、nameof 操作符、空条件运算符、字符串插入、字典初始化器、异常过滤器以及 catch 中等待。C# 7 的新特性是什么?

### 0.3.1 数字分隔符

数字分隔符使代码更具可读性。在声明变量时可以给单独的数字添加_。编译器只是删除_。下面的代码片段在 C# 7 中看起来更具可读性:

C# 6

```
long n1 = 0x1234567890ABCDEF;
```

C# 7

```
long n2 = 0x_1234_5678_90AB_CDEF;
```

在 C# 7.2 中，也可以把“_”放在开头。

C# 7.2

```
long n2 = 0x_1234_5678_90AB_CDEF;
```

数字分隔符在第 2 章介绍。

### 0.3.2　二进制字面值

C# 7 为二进制提供了一个新的字面值。二进制的值只能是 0 和 1。现在数字分隔符变得尤为重要：

C# 7

```
uint binary1 = 0b1111_0000_1010_0101_1111_0000_1010_0101;
```

二进制字面值在第 2 章介绍。

### 0.3.3　表达式体的成员

C# 6 允许使用表达式体的方法和属性。在 C# 7 中，表达式体可以与构造函数、析构函数、本地函数、属性访问器等一起使用。这里可以看到属性访问器在 C# 6 和 C# 7 之间的区别：

C# 6

```
private string _firstName;
public string FirstName
{
    get { return _firstName; }
    set { Set(ref _firstName, value); }
}
```

C# 7

```
private string _firstName;
public string FirstName
{
    get => _firstName;
    set => Set(ref _firstName, value);
}
```

表达式体的成员在第 3 章介绍。

### 0.3.4　out 变量

在 C# 7 之前，out 变量必须在使用之前声明。而在 C# 7 中，代码减少了一行，因为变量可以在使用时声明：

C# 6

```
string n = "42";
int result;
if (string.TryParse(n, out result))
{
  Console.WriteLine($"Converting to a number was successful: {result}"); }
```

C# 7

```
string n = "42";
if (string.TryParse(n, out var result))
{
    Console.WriteLine($"Converting to a number was successful: {result}");
}
```

这个特性在第 3 章介绍。

### 0.3.5　不拖尾的命名参数

C#支持可选参数需要的命名参数，但在任何情况下都可以支持可读性。C# 7.2 支持不拖尾的命名参数。参

数名可以添加到 C# 7.2 的任何参数中：

C# 7.0

```
if (Enum.TryParse(weekdayRecommendation.Entity, ignoreCase: true,
result: out DayOfWeek weekday))
{
    reservation.Weekday = weekday;
}
```

C# 7.2

```
if (Enum.TryParse(weekdayRecommendation.Entity, ignoreCase: true,
out DayOfWeek weekday))
{
    reservation.Weekday = weekday;
}
```

命名参数在第 3 章介绍。

### 0.3.6 只读结构

结构应该是只读的(有一些例外)。在 C# 7.2 中，可以使用 readonly 修饰符声明结构体，因此编译器可以验证结构体没有更改。编译器也可以使用此保证，不复制作为参数传递给它的结构，但把它传递为引用：

C# 7.2

```
public readonly struct Dimensions
{
    public double Length { get; }
    public double Width { get; }

      public Dimensions(double length, double width)
      {
         Length = length;
         Width = width;
      }

      public double Diagonal => Math.Sqrt(Length * Length + Width * Width);
}
```

只读结构在第 3 章介绍。

### 0.3.7 in 参数

C# 7.2 还允许给参数使用 in 修饰符。这就保证了所传递的值类型不会更改，并可以通过引用来传递，以避免复制：

C# 7.2

```
static void CantChange(in AStruct s)
{
    // s can't change
}
```

ref、in 和 out 修饰符在第 3 章介绍。

### 0.3.8 Private Protected

C# 7.2 添加了一个新的访问修饰符 private protected。如果某成员用于同一程序集中的类型，或派生自类的另一个程序集的类型，那么访问修饰符 protected internal 允许访问它。使用 private protected 时，上述两个条件使用 AND 而不是 OR——只有当类派生自基类，且位于同一程序集中时，才允许访问。

访问修饰符在第 4 章中介绍。

### 0.3.9 目标类型的 default

在 C# 7.1 中，定义了 default 字面值，与 default 操作符相比，它允许使用更短的语法。default 操作符总是需要类型的重复，现在不再需要了。这适用于复杂类型：

**C# 7.0**

```
int x = default(int);
ImmutableArray<int> arr = default(ImmutableArray<int>);
```

**C# 7.1**

```
int x = default;
ImmutableArray<int> arr = default;
```

default 字面值在第 5 章介绍。

### 0.3.10 本地函数

在 C# 7 之前，不可能在方法中声明函数。而可以创建一个 lambda 表达式并调用它，如 C# 6 代码片段所示：

**C# 6**

```
public void SomeFunStuff()
{
   Func<int, int, int> add = (x, y) => x + y;

   int result = add(38, 4);
   Console.WriteLine(result);
}
```

在 C# 7 中，可以在方法中声明一个本地函数。本地函数只能在方法的作用域内访问：

**C# 7**

```
public void SomeFunStuff()
{
   int add(int x, int y) => x + y;

   int result = add(38, 4);
   Console.WriteLine(result);
}
```

本地函数在第 13 章解释。本书的几个章节介绍了它的不同用途。

### 0.3.11 元组

元组允许组合不同类型的对象。在 C# 7 之前，元组是.NET Framework 中的 Tuple 类。可以使用 Item1、Item2、Item3 等访问元组的成员。在 C# 7 中，元组是该语言的一部分，可以定义成员的名称:

**C# 6**

```
var t1 = Tuple.Create(42, "astring");
int i1 = t1.Item1;
string s1 = t1.Item2;
```

**C# 7**

```
var t1 = (n: 42, s: "magic");
int i1 = t1.n;
string s1 = t1.s;
```

除此之外，新的元组是值类型(ValueTuple)，而 Tuple 类型是引用类型。元组的所有更改都包含在第 13 章中。

### 0.3.12 推断的元组名

C# 7.1 通过自动推断元组名称来扩展元组，类似于匿名类型。在 C# 7.0 中，元组的成员总是需要命名。如果元组成员的名称应该与分配给它的属性或字段相同，那么在 C# 7.1 中，如果不提供名称，它就与分配给它的成员的名称相同:

C# 7.0

```
var t1 = (FirstName: racer.FirstName, Wins: racer.Wins);
int wins = t1.Wins;
```

C# 7.1

```
var t1 = (racer.FirstName, racer.Wins);
int wins = t1.Wins;
```

### 0.3.13 拆解

不，这不是打印错误。拆解不是析构函数。元组可以拆解成独立的变量，例如：

C# 7

```
(int n, string s) = (42, "magic");
```

如果定义了 Deconstruct 方法，也可以拆解 Person 对象：

C# 7

```
var p1 = new Person("Tom", "Turbo");
(string firstName, string lastName) = p1;
```

拆解在第 13 章介绍。

### 0.3.14 模式匹配

使用模式匹配，is 操作符和 switch 语句增强为三种模式：const 模式、类型模式和 var 模式。下面的代码片段显示了使用 is 操作符的模式。第一个检查匹配常数 42，第二个检查 Person 对象，第三个用 var 模式检查每个对象。使用类型和 var 模式，可以为强类型访问声明一个变量：

C# 7

```
public void PatternMatchingWithIsOperator(object o)
{
   if (o is 42)
   {
   }
   if (o is Person p)
   {
   }
   if (o is var v1)
   {
   }
}
```

通过 switch 语句，可以对 case 子句使用相同的模式。如果模式匹配，则可以将变量声明为强类型。也可以在以下条件下使用 when 过滤模式:

C# 7

```
public void PatternMatchingWithSwitchStatement(object o)
{
   switch (o)
   {
      case 42:
      break;
```

```
        case Person p when p.FirstName == "Katharina":
        break;
        case Person p:
        break;
        case var v:
        break;
    }
}
```

模式匹配在第 13 章中介绍。

### 0.3.15 Throw 表达式

抛出异常只有在语句中才可行，在表达式中是不可行的。因此，当使用构造函数接收参数时，需要对 null 进行额外的检查，才能抛出 ArgumentNullException 异常。在 C# 7 中，可以在表达式中抛出异常，因此可以使用合并操作符在左侧为 null 时抛出 ArgumentNullException 异常。

C# 6

```
private readonly IBooksService _booksService;
public BookController(BooksService booksService)
{
    if (booksService == null)
    {
        throw new ArgumentNullException(nameof(b));
    }
    _booksService = booksService;
}
```

C# 7

```
private readonly IBooksService _booksService;
public BookController(BooksService booksService)
{
    _booksService = booksService ?? throw new ArgumentNullException(nameof(b));
}
```

Throw 表达式在第 14 章中介绍。

### 0.3.16 异步 Main()方法

在 C# 7.1 之前，Main()方法总是需要声明为类型 void。在 C# 7.1 中，Main()方法也可以是 Task 类型，使用 async 和 await 关键字：

C# 7.0

```
static void Main()
{
    SomeMethodAsync().Wait();
}
```

C# 7.1

```
async static Task Main()
{
    await SomeMethodAsync();
}
```

异步编程在第 15 章介绍。

### 0.3.17 引用语义

.NET Core 主要关注性能的提高。为引用语义添加 C#特性有助于提高性能。在 C# 7 之前，ref 关键字可以与参数一起使用，通过引用传递值类型。现在也可以对返回类型和本地变量使用 ref 关键字。

下面的代码片段声明方法 GetNumber，以返回对 int 的引用。这样，调用者可以直接访问数组中的元素，并可以更改其内容:

C# 7.0

```
int[] _numbers = { 3, 7, 11, 15, 21 };
public ref int GetNumber(int index)
{
   return ref _numbers[index];
}
```

在 C# 7.2 中，readonly 修饰符可以添加到 ref 返回值上。这样，调用者不能更改返回值的内容，但仍然使用引用语义，并可以避免在返回结果时复制值类型。调用方收到引用，但不允许更改引用：

C# 7.2

```
int[] _numbers = { 3, 7, 11, 15, 21 };
public ref readonly int GetNumber(int index)
{
   return ref _numbers[index];
}
```

在 C# 7.2 之前，C#可以创建引用类型(类)和值类型(结构)。然而，装箱时，结构体也可以存储在堆中。在 C# 7.2 中，可以声明一个类型 ref struct，该类型只允许放在堆栈上：

C# 7.2

```
ref struct OnlyOnTheStack
{
}
```

引用的新特性在第 17 章中介绍。

## 0.4 ASP.NET Core 中的新特性

在.NET Core 和 Visual Studio 2017 中，有一个新的项目文件。在 Visual Studio 2015 中是预览版本的.NET Core 工具，在 Visual Studio 2017 中已经发布了。该工具使用 csproj 文件切换到 MSBuild 环境，所以现在有了 csproj 文件用于.NET Framework 和.NET Core 应用程序。然而，这并不是前几代的 csproj 文件。现在的 csproj 文件要短得多，简化了许多，也可以使用简单的文本编辑器来修改它们。

.NET Core 2.0 通过.NET Standard 2.0 中定义的类和方法得到增强，这更便于将现有的.NET Framework 应用程序迁移到.NET Core。

创建 ASP.NET Core 项目，不仅简化了 csproj 文件，而且简化了 C#源代码。使用默认的 WebHostBuilder 时，会预定义更多的内容。添加配置和日志提供程序时，不需要自己添加它们。在 ASP.NET Core MVC 中，有了一些小的改进——例如，视图组件现在可以在标记辅助程序中使用。

还有一种新的技术——Razor 页面，比 ASP.NET Core MVC 更容易学习。有些应用程序不需要来自模型-视图-控制器模式的抽象，此时就可以使用 Razor 页面。

## 0.5 UWP 的新特性

Windows 10 一年更新两次(如果在 Windows Insiders 程序中，就会更频繁地得到更新，但这对大多数用户来说并不正常)。每次 Windows 更新都会发布一个新的 SDK。最新的两项更新是 Creators Update(构建号 15063, 2017 年 3 月)和 Fall Creators Update (构建号 16299, 2017 年 10 月)。

微软继续提供集成在 Windows 控件中的新设计特性。新的设计命名为 Fluent Design，它集成到标准控件中，也可以直接访问——例如，使用 acrylic 和 reveal brushes。在应用程序中通过 ParallaxView 添加了视差效果。

添加特性也提高了生产力。可以使用 Windows Template Studio (Visual Studio 的一个扩展)中的模板编辑器创建多个页面，并使用预先生成的服务。

XAML 通过有条件的 XAML 进行了增强，使其更容易支持多个 Windows 10 版本，但使用旧版本 Windows 10 中没有的新功能。

InkCanvas 控件提供了新的标尺，可以轻松地集成到应用程序中。NavigationView 可以很容易地创建带有汉堡包按钮和 SplitView 的自适应菜单。本书的第Ⅳ部分将详细介绍所有这些新特性以及更多的内容。

## 0.6 编写和运行 C#代码的环境

.NET Core 运行在 Windows、Linux 和 Mac 操作系统上。使用 Visual Studio Code(https://code. visualstudio.com)，可以在任何操作系统上创建和构建程序。

最好用的开发工具是 Visual Studio 2017，也是本书使用的工具。可以使用 Visual Studio Community 2017 版(https://www.visualstudio.com)，但本书介绍的一些功能只有 Visual Studio 的企业版提供。需要企业版时会提到。Visual Studio 2017 需要 Windows 10 构建号 1507 或更高版本，要求使用 Windows 8.1、Windows Server 2012 R2 或 Windows 7 SP1。要构建和运行本书中的 Windows 应用程序(UWP)，需要 Windows 10。

要为 iOS 创建和构建 Xamarin 应用程序，还需要用于构建系统的 Mac。没有 Mac，仍然可以为 Windows 和 Android 开发 Xamarin 应用程序。

在Mac上开发应用程序，可以使用Visual Studio for Mac：https://www.visualstudio.com/vs/Visual - Studio - Mac/。可以使用这个工具创建ASP.NET Core和Xamarin应用程序，但是不能创建和测试Windows应用程序。

## 0.7 本书内容

本书首先在第 1 章介绍.NET 的整体体系结构，给出编写托管代码所需要的背景知识。此后概述不同的应用程序类型，学习如何用新的开发环境 CLI 编译程序，介绍在 Visual Studio 中开始学习的最重要部分。之后本书分几部分介绍 C#语言及其在各个领域中的应用。

### 第 I 部分——C#语言

该部分介绍 C#语言的良好背景知识。尽管这一部分假定读者是有经验的编程人员，但它没有假设读者拥有任何特定语言的知识。首先介绍 C#的基本语法和数据类型，再介绍 C#的面向对象特性，之后介绍 C#中的一些高级编程主题，如委托、lambda 表达式和语言集成查询(Language Integrated Query，LINQ)。

由于 C#包含许多函数式编程的特性，因此需要了解元组和模式匹配的函数式编程基础，讨论异步编程和引用语义的新语言特性。本部分最后探讨 Visual Studio 2017 的许多特性，还将了解 Docker 的基础知识，以及 Visual Studio 2017 支持 Docker 的方式。

### 第 II 部分——.NET Core 与 Windows Runtime

第 19～29 章介绍独立于应用程序类型的.NET Core 和 Windows Runtime(运行库)。本部分在第 19 章中介绍如何创建库和 NuGet 包，学习如何以最好的方式使用.NET 标准。

依赖注入(Dependency Injection，DI)与.NET Core 一起使用：服务注入 Entity Framework Core 和 ASP.NET Core。ASP.NET Core MVC 使用了数百个服务。DI 便于跨 WPF、UWP 和 Xamarin 使用相同的代码。第 20 章专门介绍 DI 的基础，还可以在 DI 容器 Microsoft.Extensions .DependencyInjection 中学到高级特性，包括调整非微软容器。其他许多章节也使用 DI。

第 21 章介绍使用任务并行库(Task Parallel Library，TPL)进行并行编程，以及用于同步的各种对象。

第 22 章学习如何访问文件系统，读取文件和目录，了解如何使用 System.IO 名称空间中的流和 Windows 运行库中的流来编写 Windows 应用程序。

第 23 章学习使用套接字和使用更高级别的抽象(如 HttpClient)的联网的核心基础。

第 24 章利用流来了解安全性，以及如何加密数据，允许进行安全的转换。本章还讨论了创建 Web 应用程序时需要了解的一些主题，如 SQL 注入和跨站点请求伪造攻击的问题。

第 25 和 26 章展示了如何访问数据库。第 25 章使用 ADO.NET 直接解释事务，并涵盖了使用.NET 核心的环境事务。第 26 章介绍了 Entity Framework Core 2.0 提供的所有新特性。EF Core 2.0 有旧的 Entity Framework 6.x

无法提供的许多特性。

第 27 章介绍使用本地化的技术本地化应用程序，该技术对 Windows 和 Web 应用程序都非常重要。

用 C#代码创建功能时，不要跳过创建单元测试的步骤。一开始需要更多的时间，但随着时间的推移，添加功能和维护代码时，就会看到其优势。第 28 章介绍如何创建单元测试，包括 Visual Studio 2017 中的实时单元测试、网络测试和编码的 UI 测试。

第 29 章介绍了.NET Core 的日志功能，以及使用 Visual Studio AppCenter 获取分析信息。

**第 III 部分——Web 应用程序和服务**

本部分将介绍 Web 应用程序和服务。本部分从第 30 章开始，提供了 ASP.NET Core 的基础。使用 MVC 模式创建 Web 应用程序，包括新技术 Razor 页面，在第 31 章中介绍。第 32 章涵盖了 ASP.NET Core 的 REST 服务特性：Web API。

**第 IV 部分——应用程序**

本部分介绍如何使用 XAML 构建应用程序——Universal Windows 应用程序和 Xamarin。了解 Windows 应用程序的基础，包括第 33 章中 XAML 的基础，其中包含 XAML 语法、依赖属性和标记扩展，可以在其中创建自己的 XAML 语法。本章介绍了 Windows 控件的不同类别以及与 XAML 绑定数据的基础。

第 34 章主要关注 MVVM(模型-视图-视图模型)模式。该章将学习如何利用基于 XAML 的应用程序的数据绑定特性，这些特性允许在 Windows 应用程序、WPF 和 Xamarin 之间共享大量代码。也可以分享许多为 iOS 和 Android 平台开发的代码。创建 WPF 应用程序并没有在本书中介绍——这一技术在最近几年并没有得到多少改进，应该考虑转向通用 Windows 平台，如果使用第 34 章学到的知识，就更容易实现这一点。WPF 应用程序仍然需要维护。要想更深入地了解 WPF，应该阅读本书的上一版。

第 35 章介绍基于 XAML 的应用程序的样式化。第 36 章介绍了用通用 Windows 平台创建 Windows 应用程序的高级功能。展示了应用程序服务、上墨(inking)、AutoSuggest 控件、高级编译绑定特性等。

第 37 章帮助启动 Windows、Android 和 iPhone 的 Xamarin 开发，并展示幕后发生的事情。学习 Xamarin.Android、Xamarin.iOS 和 Xamarin.Forms 之间的不同之处。了解 Xamarin.Forms 控件与 Windows 控件的不同之处，更快地从 Windows 开发转向 Xamarin。本章的较大示例使用与第 34 章中的 Windows 应用程序相同的 MVVM 库。

**附加的章节**

附加的第 1 章讨论了 Microsoft Composition，它允许创建容器和部件之间的独立性。附加的第 2 章论述如何将对象序列化到 XML 和 JSON 中，以及用于读取和编写 XML 的不同技术。

Web 应用程序的发布和订阅技术使用 ASP.NET Core 技术 WebHooks 和 SignalR 的形式，在附加的第 3 章中讨论。附加的第 4 章对使用 Bot 服务和 Azure 认知服务创建应用程序有了新的认识。

附加的第 5 章涵盖了一些与 Windows 应用程序相关的额外主题：使用相机、地理定位来访问当前的位置信息，以各种格式显示地图的 MapControl，以及几个传感器(比如提供光线信息和测量重力的传感器)。

可以扫描封底二维码查看附加的 5 章内容。

## 0.8 如何下载本书的示例代码

在读者学习本书中的示例时，可以手工输入所有的代码，也可以使用本书附带的源代码文件。本书使用的所有源代码都可以从本书合作站点 http://www.wrox.com/上下载。登录到站点 http://www.wrox.com/，使用 Search 工具或书名列表就可以找到本书。接着单击本书细目页面上的 Download Code 链接，就可以获得所有的源代码，也可以扫描封底的二维码获取本书的源代码。

**注意：**
许多图书的书名都很相似，所以通过 ISBN 查找本书是最简单的，本书英文版的 ISBN 是 978-1-119-44927-0。

在下载代码后，只需要用自己喜欢的解压缩软件对它进行解压缩即可。另外，也可以进入 http://www.wrox.com/dynamic/books/download.aspx 上的 Wrox 代码下载主页，查看本书和其他 Wrox 图书的所有代码。

## 0.9 GitHub

源代码也可以在 GitHub 上提供，网址是 https://www.github.com/ProfessionalCSharp/ ProfessionalCSharp7。在 GitHub 中，还可以使用 Web 浏览器打开每个源代码文件。使用这个网站时，可以把完整的源代码下载到一个 zip 文件。还可以将源代码复制到系统上的本地目录。只需要安装 git 工具，为此可以使用 Visual Studio 或者从 https://git-scm.com/downloads 下载 Windows、Linux 和 Mac 的 git 工具。要将源代码复制到本地目录，请使用 git clone：

```
> git clone https://www.github.com/ProfessionalCSharp/ProfessionalCSharp7
```

使用此命令，把完整的源代码复制到子目录 ProfessionalCSharp7。之后就可以开始处理源文件了。

Visual Studio 的更新可用，SignalR 库发布后，源代码将在 GitHub 上更新。如果在复制源代码之后源代码发生了变化，就可以在将当前目录更改为源代码目录后，提取最新的更改：

```
> git pull
```

如果对源代码做了一些更改，git pull 可能会导致错误。如果发生这种情况，可以把更改隐藏起来，然后再取出来：

```
> git stash
> git pull
```

git 命令的完整列表可以在 https://git-scm.com/docs 上找到。

如果源代码有问题，可以在存储库中报告问题。在浏览器中打开 https://github.com/ProfessionalCSharp/ ProfessionalCSharp7，单击 Issues 选项卡，单击 New Issue 按钮。这将打开一个编辑器，如图 1 所示，尽可能详细地描述问题。

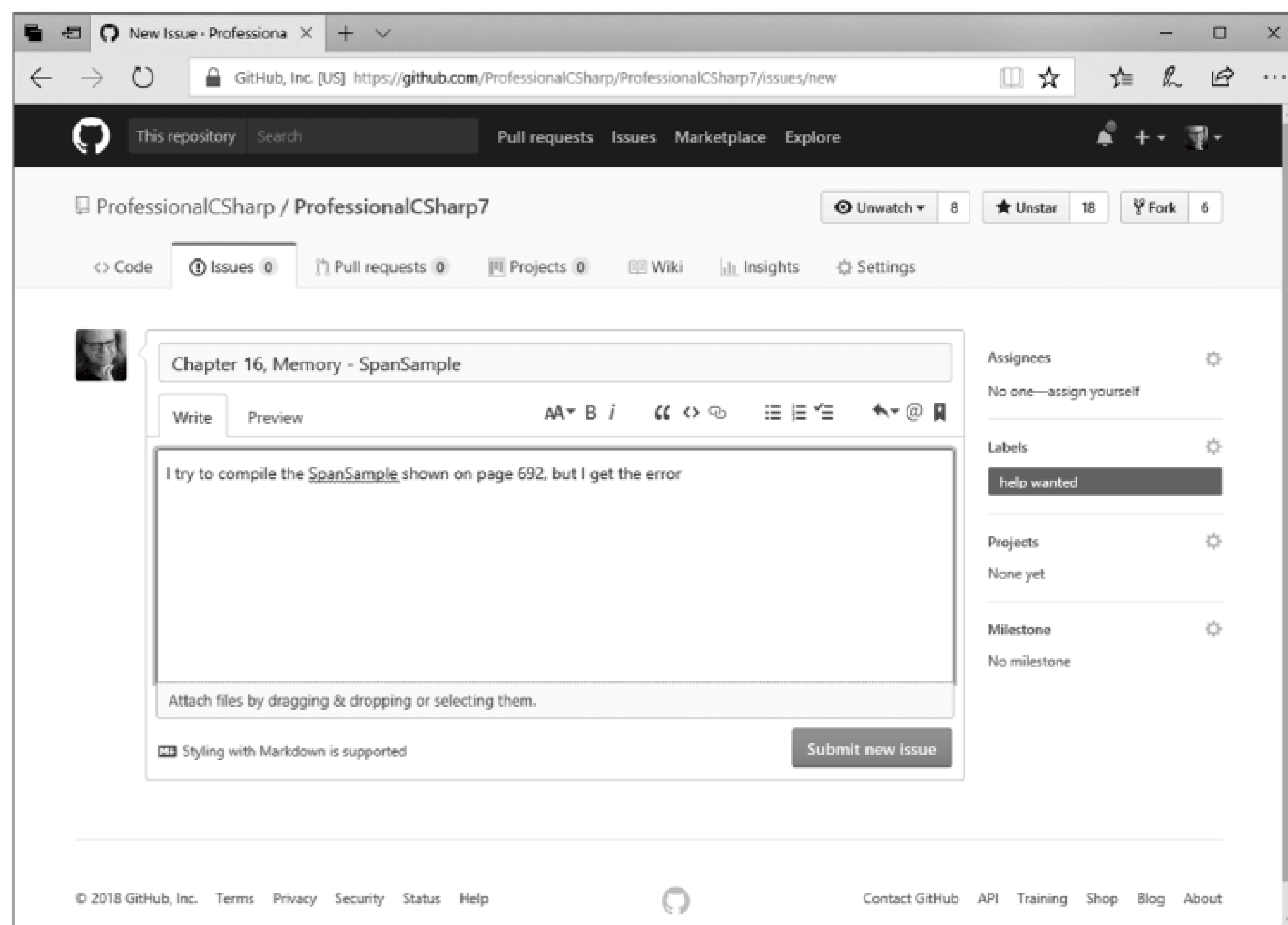



图 1

为了报告问题，需要一个 GitHub 账户。如果有一个 GitHub 账户，也可以将源代码存储库分叉到账户上。有关使用 GitHub 的更多信息，请查看 https://guides .github.com/activities/hello-world。

**注意：**

可以读取源代码和相关问题，并在不加入 GitHub 的情况下在本地复制存储库。要在 GitHub 上发布问题并创建自己的存储库，需要自己的 GitHub 账户。

## 0.10 勘误表

尽管我们已经尽力保证不出现错误，但错误总是难免的，如果你在本书中找到了错误，如拼写错误或代码错误，请告诉我们，我们将非常感激。通过勘误表，可以让其他读者避免受挫，当然，这还有助于提供更高质量的信息。

要在网站上找到本书的勘误表，可以登录 http://www.wrox.com，通过 Search 工具或书名列表查找本书，然后在本书的细目页面上，单击 Book Errata 链接。在这个页面上可以查看 Wrox 编辑已提交和粘贴的所有勘误项。完整的图书列表还包括每本书的勘误表，网址是 www.wrox.com/misc-pages/ booklist.shtml。

如果没有在 Book Errata 页面上发现自己找到的错误，请访问 www.wrox.com/contact/techsupport.shtml，填好表单，将你找到的错误发送给我们。我们将检查信息，如果合适的话，将消息发送到该书的勘误页面，并在该书的后续版本中修复问题。

# 目　录

## 第 I 部分　C# 语 言

## 第 II 部分 .NET Core 与 Windows Runtime

## 第 III 部分 Web 应用程序和服务

## 第 IV 部分 应用程序

**附赠章节电子版(请扫描封底二维码获取)**

第 I 部分

# C# 语 言

- 第 1 章　.NET 应用程序和工具
- 第 2 章　核心 C#
- 第 3 章　对象和类型
- 第 4 章　继承
- 第 5 章　泛型
- 第 6 章　运算符和类型强制转换
- 第 7 章　数组
- 第 8 章　委托、lambda 表达式和事件
- 第 9 章　字符串和正则表达式
- 第 10 章　集合
- 第 11 章　特殊的集合
- 第 12 章　LINQ
- 第 13 章　C#函数式编程
- 第 14 章　错误和异常

- 第 15 章　异步编程
- 第 16 章　反射、元数据和动态编程
- 第 17 章　托管和非托管的资源
- 第 18 章　Visual Studio 2017

# 第 1 章

# .NET 应用程序和工具

**本章要点**

- 回顾.NET 的历史
- 理解.NET Framework 和.NET Core 之间的差异
- NuGet 包
- 公共语言运行库
- Windows 运行库的特性
- 编写“Hello World!”程序
- .NET Core 命令行界面
- Visual Studio 2017
- 通用 Windows 平台
- 创建 Windows 应用程序的技术
- 创建 Web 应用程序的技术

**本章源代码下载：**

单击 www.wrox.com 的 Download Code 选项卡可下载本章源代码。源代码也可以在 HelloWorld 目录的 https://github.com/ProfessionalCSharp/ProfessionalCSharp7 中找到。

本章代码分为以下几个主要的示例文件：

- HelloWorld
- WebApp
- SelfContained HelloWorld

## 1.1 选择技术

.NET 是在 Windows 平台上创建应用程序的杰出技术。但现在，.NET 是在 Windows、Linux 和 Mac 上创建应用程序的杰出技术。

.NET Core 是.NET 自其发明以来最大的变化。目前.NET 代码是开源的代码，还可以为其他平台创建应用程序。.NET 使用现代模式。.NET Core 和 NuGet 包允许微软公司以更短的更新周期提供新特性。应该使用什么

技术创建应用程序并不容易决定，本章将提供这方面的帮助。其中包含用于创建 Windows、Web 应用程序和服务的不同技术的信息，指导选择什么技术进行数据库访问，凸显了.NET Framework 和.NET Core 之间的差异。

## 1.2　回顾.NET 历史

要更好地理解.NET 和 C#的可用功能，最好先了解它的历史。表 1-1 显示了.NET Framework 的版本、对应的公共语言运行库(Common Language Runtime，CLR)的版本、C#的版本以及 Visual Studio 的版本，并指出相应版本的发布年份。除了知道使用什么技术之外，最好也知道不推荐使用什么技术，因为这些技术会被替代。

表 1-1

| .NET Framework | CLR | C# | Visual Studio |
|---|---|---|---|
| 1.0 | 1.0 | 1.0 | 2002 |
| 1.1 | 1.1 | 1.2 | 2003 |
| 2.0 | 2.0 | 2.0 | 2005 |
| 3.0 | 2.0 | 2.0 | 2005+扩展版 |
| 3.5 | 2.0 | 3.0 | 2008 |
| 4.0 | 4.0 | 4.0 | 2010 |
| 4.5 | 4.0 | 5.0 | 2012 |
| 4.5.1 | 4.0 | 5.0 | 2013 |
| 4.6 | 4.0 | 6 | 2015 |
| 4.7 | 4.0 | 7 | 2017 |

当使用.NET Core 创建应用程序时，了解支持级别的时间框架非常重要。LTS(Long Time Support，长时间支持)的支持长度比 Current 长，但 Current 会更快地获得新特性。LTS 在发行后或下一个 LTS 版本发布的 12 个月后得到 3 年的支持，以较短的版本为准。因此，如果下一个 LTS 版本在 2018 年 6 月 27 日之前没有发布，那么.NET Core 1.0 将支持到 2019 年 6 月 27 日。如果下一个 LTS 版本在更早的时候发布，那么在下一个 LTS 发布后的一年，.NET Core 1.0 将得到支持。

.NET Core 1.1 最初是一个 Current 版本，但它变成了 LTS，它得到的支持长度与.NET Core 1.0 相同。

.NET Core 2.0 是一个 Current 支持级别的版本。这意味着它将得到 3 年的支持，下一个 LTS 发布后的 12 个月，或发布下一个 Current 版本后的 3 个月——以较短的时间为准。可以假定，最后一个选项是视情况而定，.NET Core 2.0 将在.NET Core 2.1 可用后 3 个月得到支持。

表 1-2 列出了.NET Core 版本、发布日期和支持级别。

表 1-2

| .NET Core 版本 | 发 布 日 期 | 支 持 级 别 |
|---|---|---|
| 1.0 | 2016 年 6 月 27 日 | LTS |
| 1.1 | 2016 年 11 月 16 日 | LTS* |
| 2.0 | 2017 年 8 月 14 日 | Current |

下面各小节详细介绍这两个表，以及 C#和.NET 的发展。

### 1.2.1　C# 1.0 —— 一种新语言

C# 1.0 是一种全新的编程语言，用于.NET Framework。开发它时，.NET Framework 由大约 3000 个类和 CLR 组成。

创建 Java 的 Sun 公司申请法庭判决不允许微软公司更改 Java 代码后，Anders Hejlsberg 设计了 C#。Hejlsberg 为微软公司工作之前，在 Borland 公司设计了 Delphi 编程语言(一种 Object Pascal 语言)。Hejlsberg 在微软公司负责 J++(Java 编程语言的微软版本)。鉴于 Hejlsberg 的背景，C#编程语言主要受到 C++、Java 和 Pascal 的影响。

因为 C#的创建晚于 Java 和 C++，所以微软公司分析了其他语言中典型的编程错误，完成了一些不同的工作来避免这些错误。这些不同的工作包括：

- 在 if 语句中，布尔(Boolean)表达式是必需的(C++也允许在这里使用整数值)。
- 允许使用 struct 和 class 关键字创建值类型和引用类型(Java 只允许创建自定义引用类型；在 C++中，struct 和 class 之间的区别只是访问修饰符的默认值不同)。
- 允许使用虚拟方法和非虚拟方法 (这类似于 C++；Java 总是创建虚拟方法)。

当然，阅读本书，会看到更多的变化。

现在，C#是一种纯粹的面向对象编程语言，具备继承、封装和多态性等特性。C#也提供了基于组件的编程改进，如委托和事件。

在.NET 和 CLR 推出之前，每种编程语言都有自己的运行库。在 C++中，C++运行库与每个 C++程序链接起来。Visual Basic 6 有自己的运行库 VBRun。Java 的运行库是 Java 虚拟机(Java Virtual Machine，JVC)——可以与 CLR 相媲美。CLR 是每种.NET 编程语言都使用的运行库。推出 CLR 时，微软公司提供了 JScript .NET、Visual Basic .NET、 Managed C++ 和 C#。JScript .NET 是微软公司的 JavaScript 编译器，与 CLR 和.NET 类一起使用。Visual Basic .NET 是提供.NET 支持的 Visual Basic，现在再次简称为 Visual Basic。Managed C++是混合了本地 C++代码与 Managed .NET 代码的语言。今天与.NET 一起使用的新 C++语言是 C++/ CLR。

.NET 编程语言的编译器生成中间语言(Intermediate Language，IL)代码。IL 代码看起来像面向对象的机器码，使用工具 ildasm.exe 可以打开包含.NET 代码的 DLL 或 EXE 文件来检查 IL 代码。CLR 包含一个即时(Just-In-Time，JIT)编译器，当程序开始运行时，JIT 编译器会从 IL 代码中生成本地代码。

> **注意：**
> IL 代码也称为托管代码。

CLR 的其他部分是垃圾收集器(GC)、调试器扩展和线程实用工具。垃圾收集器负责清理不再引用的托管内存，这个安全机制使用代码访问安全性来验证允许代码做什么。调试器扩展允许在不同的编程语言之间启动调试会话 (例如，在 Visual Basic 中启动调试会话，在 C#库内继续调试)。线程实用工具负责在底层平台上创建线程。

.NET Framework 的第 1 版已经很大了。类在名称空间内组织，以便于导航可用的 3000 个类。使用名称空间组织类，允许在不同的名称空间中有相同的类名，以解决冲突。.NET Framework 的第 1 版允许使用 Windows Forms(名称空间 System.Windows.Forms)创建 Windows 桌面应用程序，使用 ASP.NET Web Forms (System.Web)创建 Web 应用程序，使用 ASP.NET Web Services 与应用程序和 Web 服务通信，使用.NET Remoting 在.NET 应用程序之间更迅速地通信，使用 Enterprise Services 创建运行在应用程序服务器上的 COM +组件。

ASP.NET Web Forms 是创建 Web 应用程序的技术，其目标是开发人员不需要了解 HTML 和 JavaScript。服务器端控件会创建 HTML 和 JavaScript，这些控件的工作方式类似于 Windows Forms 本身。

C# 1.2 和.NET 1.1 主要是错误修复版本，改进较小。

> **注意：**
> 继承在第 4 章中讨论，委托和事件在第 8 章中讨论。

> **注意：**
> .NET 的每个新版本都有 Professional C#图书的新版本。对于.NET 1.0，这本书已经是第 2 版了，因为第 1 版是以.NET 1.0 的 Beta 2 为基础出版的。目前，本书是第 11 版。

### 1.2.2 带有泛型的 C# 2 和.NET 2

C# 2 和.NET 2 是一个巨大的更新。在这个版本中，改变了 C#编程语言，建立了 IL 代码，所以需要新的 CLR 来支持 IL 代码的增加。一个大的变化是泛型。泛型允许创建类型，而不需要知道使用什么内部类型。所使用的内部类型在实例化(即创建实例)时定义。

C#编程语言中的这个改进也导致了 Framework 中多了许多新类型，例如 System.Collections.Generic 名称空间中新的泛型集合类。有了这个类，1.0 版本定义的旧集合类就很少用在新应用程序中了。当然，旧类现在仍然在工作，甚至在新的.NET Core 版本中也是如此。

> **注意：**
> 本书一直在使用泛型，详见第 5 章。第 10 章介绍了泛型集合类。

### 1.2.3 .NET 3.0 ——Windows Presentation Foundation

发布.NET 3.0 时，不需要新版本的 C#。3.0 版本只提供了新的库，但它发布了大量新的类型和名称空间。Windows Presentation Foundation(WPF)可能是新框架最大的一部分，用于创建 Windows 桌面应用程序。Windows Forms 包括本地 Windows 控件，且基于像素；而 WPF 基于 DirectX，独立绘制每个控件。WPF 中的矢量图形允许无缝地调整任何窗体的大小。WPF 中的模板还允许完全自定义外观。例如，用于苏黎世机场的应用程序可以包含看起来像一架飞机的按钮。因此，应用程序的外观可以与之前开发的传统 Windows 应用程序不同。System.Windows 名称空间下的所有内容都属于 WPF，但 System.Windows.Forms 除外。有了 WPF，用户界面可以使用 XML 语法 XAML(XML for Applications Markup Language)设计。

.NET 3.0 推出之前，ASP.NET Web Services 和.NET Remoting 用于应用程序之间的通信。Message Queuing 是用于通信的另一个选择。各种技术有不同的优点和缺点，它们都用不同的 API 进行编程。典型的企业应用程序必须使用一个以上的通信 API，因此必须学习其中的几项技术。WCF(Windows Communication Foundation) 解决了这个问题。WCF 把其他 API 的所有选项结合到一个 API 中。然而，为了支持 WCF 提供的所有功能，需要配置 WCF。

.NET 3.0 版本的第三大部分是 Windows WF(Workflow Foundation)和名称空间 System.Workflow。微软公司不是为几个不同的应用程序创建自定义的工作流引擎(微软公司本身为不同的产品创建了几个工作流引擎)，而是把工作流引擎用作.NET 的一部分。

有了.NET 3.0，Framework 的类从.NET 2.0 的 8 000 个增加到约 12 000 个。

> **注意：**
> 要学习 WPF 和 WCF，需要阅读本书的前一版本《C#高级编程(第 10 版) C# 6 & .NET Core 1.0》。

### 1.2.4 C# 3 和.NET 3.5——LINQ

.NET 3.5 和新版本 C# 3 一起发布。主要改进是使用 C#定义的查询语法，它允许使用相同的语法来过滤和排序对象列表、XML 文件和数据库。语言增强不需要对 IL 代码进行任何改变，因为这里使用的 C#特性只是语法糖。所有的增强也可以用旧的语法实现，只是需要编写更多的代码。C#语言很容易进行这些查询。有了 LINQ 和 lambda 表达式，就可以使用相同的查询语法来访问对象集合、数据库和 XML 文件。

为了访问数据库并创建 LINQ 查询，LINQ to SQL 发布为.NET 3.5 的一部分。在.NET 3.5 的第一个更新中，发布了 Entity Framework 的第一个版本。LINQ to SQL 和 Entity Framework 都提供了从层次结构到数据库关系的映射和 LINQ 提供程序。Entity Framework 更强大，但 LINQ to SQL 更简单。随着时间的推移，LINQ to SQL 的特性在 Entity Framework 中实现了，并且 Entity Framework 会一直保留这些特性。Entity Framework 的新版本 Entity Framework Core (EF Core)看起来与第一版非常不同。

另一种引入为.NET 3.5 一部分的技术是 System.AddIn 名称空间，它提供了插件模型。这个模型提供了甚至在过程外部运行插件的强大功能，但它使用起来也很复杂。

> **注意：**
> LINQ 详见第 12 章，Entity Framework 的最新版本与.NET 3.5 版本有很大差别，参见第 26 章。

### 1.2.5　C# 4 和.NET 4.0——dynamic 和 TPL

C# 4 的主题是动态集成脚本语言，使其更容易使用 COM 集成。C#语法扩展为使用 dynamic 关键字、命名参数和可选参数，以及用泛型增强的协变和逆变。

其他改进在.NET Framework 中进行。有了多核 CPU，并行编程就变得越来越重要。任务并行库(Task Parallel Library，TPL)使用 Task 类和 Parallel 类抽象出线程，更容易创建并行运行的代码。

因为用.NET 3.0 创建的工作流引擎没有履行自己的诺言，所以全新的 Windows Workflow Foundation 成为.NET 4.0 的一部分。为了避免与旧工作流引擎冲突，新的工作流引擎是在 System.Activity 名称空间中定义的。

C# 4 的增强还需要一个新版本的运行库。运行库从版本 2 跳到版本 4。

发布 Visual Studio 2010 时，附带了一项创建 Web 应用程序的新技术：ASP.NET MVC 2.0。与 ASP.NET Web Forms 不同，该技术关注于模型-视图-控制器(MVC)模式，该模式由项目结构强制执行。这项技术也关注于编程 HTML 和 JavaScript。HTML 和 JavaScript 通过 HTML 5 的发布在开发者社区中获得了巨大的推动。由于这一技术非常新颖，而且是到 Visual Studio 的带外(Out Of Band，OOB)，因此 ASP.NET MVC 是定期更新的。

> **注意：**
> C# 4 的 dynamic 关键字参见第 16 章。任务并行库参见第 21 章。ASP.NET 的新一代 ASP.NET Core 参见第 30 章，ASP.NET Core MVC6 参见第 31 章。

### 1.2.6　C# 5 和异步编程

C# 5 只有两个新的关键字：async 和 await，然而，它大大简化了异步方法的编程。在 Windows 8 中，触摸变得更加重要，不阻塞 UI 线程也变得更加重要。用户使用鼠标，习惯于花些时间滚动屏幕。然而，在触摸界面上使用手势时，反应不及时很不好。

Windows 8 还为 Windows Store 应用程序(也称为 Modern 应用程序、Metro 应用程序、通用 Windows 应用程序，最近称为 Windows 应用程序)引入了一个新的编程接口：Windows 运行库。这是一个本地运行库，看起来像是使用语言投射的.NET。许多 WPF 控件都为新的运行库重写了，.NET Framework 的一个子集可以使用这样的应用程序。

System.AddIn 框架过于复杂、缓慢，所以用.NET 4.5 创建了一个新的合成框架：Managed Extensibility Framework 和名称空间 System.Composition。

独立于平台的通信的新版本是由 ASP.NET Web API 提供的。WCF 提供有状态和无状态的服务，以及许多不同的网络协议，而 ASP.NET Web API 则简单得多，它是基于 Representational State Transfer(REST)软件架构风格的。

> **注意：**
> C# 5 的 async 和 await 关键字在第 15 章中详细讨论，其中也介绍.NET 在不同时期使用的不同异步模式。
> MEF 参见网上附加第 1 章。Windows 应用程序参见第 33～36 章，Web API 和 ASP.NET Core MVC 参见第 32 章。

### 1.2.7 C# 6 和.NET Core 1.0

C# 6 没有由泛型、LINQ 和异步带来的巨大改进，但有许多小而实用的语言增强，可以在几个地方减少代码的长度。很多改进都通过新的编译器引擎 Roslyn 或.NET Compiler Platform 实现。

完整的.NET Framework 并不是近年来使用的唯一.NET Framework。有些场景需要较小的框架。2007 年，发布了 Microsoft Silverlight 的第一个版本(代码名为 WPF/E，即 WPF Everywhere)。Silverlight 是一个 Web 浏览器插件，支持动态内容。Silverlight 的第一个版本只支持通过 JavaScript 编程。第 2 个版本包含.NET Framework 的子集。当然，不需要服务器端库，因为 Silverlight 总是在客户端运行，但附带 Silverlight 的 Framework 也删除了核心特性中的类和方法，使其更简洁，便于移植到其他平台。用于桌面的 Silverlight 版本(第 5 版)在 2011 年 12 月发布。Silverlight 也用于 Windows Phone 的编程。Silverlight 8.1 进入 Windows Phone 8.1，但这个版本的 Silverlight 不同于桌面版本。

在 Windows 桌面上，有如此巨大的.NET 框架，需要更快的开发节奏，也需要较大的改进。在 DevOps 中，开发人员和操作员一起工作，甚至是同一个人不断地给用户提供应用程序和新特性，需要使新特性快速可用。由于框架巨大，且有许多依赖关系，创建新的特性或修复缺陷是一项不容易完成的任务。

有了几个较小的.NET 版本(如 Silverlight、用于 Windows Phone 的 Silverlight)，在.NET 的桌面版本和较小版本之间共享代码就很重要。在不同.NET 版本之间共享代码的一项技术是可移植库。随着时间的推移，有了许多不同的.NET Framework 和版本，可移植库的管理已成为一场噩梦。

为了解决所有这些问题，需要.NET 的新版本(是的，的确需要解决这些问题)。Framework 的新版本命名为.NET Core。.NET Core 较小，是开源的，带有模块化的 NuGet 包，以及分布给每个应用程序的运行库，不仅可用于 Windows 的桌面版，也可用于许多不同的 Windows 设备，以及 Linux 和 OS X。

为了创建 Web 应用程序，完全重写了 ASP.NET，得到了 ASP.NET Core 1.0。这个版本不完全向后兼容老版本，需要对现有的 ASP.NET MVC(和 ASP.NET Core MVC)代码进行一些修改。然而，与旧版本相比，它也有很多优点，例如每一个网络请求的开销较低，性能更好，也可以在 Linux 上运行。ASP.NET Web Forms 不包含在这个版本中，因为 ASP.NET Web Forms 不是专为最佳性能而设计的，它基于 Windows Forms 应用程序开发人员熟悉的模式来提高对开发人员的友好性。

当然，并不是所有应用程序都很容易改为使用.NET Core。所以这个巨大的框架也会进行改进——即使这些改进的完成速度没有.NET Core 那么快，也是要改进的。.NET Framework 完整的新版本是 4.6。ASP.NET Web Forms 的小更新包在完整的.NET 上可用。

> **注意：**
> C#语言的变化参见第 I 部分中所有的语言章节，例如，只读属性参见第 3 章，nameof 运算符和空值传播参见第 6 章，字符串插值参见第 9 章，异常过滤器参见第 14 章。

### 1.2.8 C# 7 和.NET Core 2.0

C#更新具有更快的速度。主要版本 7.0 在 2017 年 3 月发布，次级版本 7.1 和 7.2 分别在 2017 年 8 月和 12 月后不久发布。通过项目设置可以选择要使用的编译器版本。

C# 7 引入了许多新特性。这些特性中最重要的部分来自函数式编程：模式匹配和元组。

> **注意：**
> 模式匹配和元组参见第 13 章。

.NET Core 2.0 的重点是更容易将使用.NET Framework 编写的现有应用程序引入.NET Core。以前不能用于.NET Core、但仍在许多.NET Framework 应用程序和库中使用的类型，现在可以用于.NET Core。在.NET Core 2.0 中添加了两万多个 API。例如，二进制序列化和 DataSet 又回来了，还可以在 Linux 上使用这些特性。另一

个有助于将旧应用程序引入.NET Core 的特性是 Windows Compatibility Pack (Microsoft.Windows. Compatibility)。这个 NuGet 包定义了用于 WCF、注册表访问、加密、目录服务、绘图等的 API。当前状态参见 https://github.com/dotnet/designs/blob/master/accepted/compat-pack/compat-pack.md。

.NET Standard 是一个规范，它定义了在任何支持该标准的平台上应该使用哪些 API。标准版本越高，可用的 API 就越多。.NET Standard 2.0 将标准扩展了两万多个 API，并得到了.NET Framework 4.6.1、.NET Core 2.0 和通用 Windows 平台(Windows 应用程序)的支持，开始于构建版 16299 (Windows 10 的 Fall Creators Update)。

**注意：**
第 19 章详细介绍了.NET Standard。

要检查应用程序是否可以轻松地移植到.NET Core 中，可以使用.NET Portability Analyzer (.NET 可移植性分析器)。此工具可以安装为 Visual Studio 的扩展。它会分析二进制文件。还可以为希望获得的版本和框架配置可移植性信息，为.NET Core、.NET Framework、.NET Standard、Mono、Silverlight、Windows、Xamarin 等提供可移植性信息。结果可以是 JSON、HTML 和 Excel。

图 1-1 显示了在选择.NET Framework 二进制文件后的总结报告，其中，该二进制文件与.NET Framework 100%兼容，与.NET Core 96.67%兼容，与 Windows 应用程序 69.7%兼容。图 1-2 显示了有问题的 API 的详细信息。

图 1-1

图 1-2

## 1.2.9　选择技术，继续前进

知道框架内技术相互竞争的原因后，就更容易选择用于编写应用程序的技术。例如，如果创建新的 Windows 应用程序，使用 Windows Forms 就不是好的解决方案，而应该使用基于 XAML 的技术，例如 Universal Windows Platform (UWP)。当然，使用其他技术仍然有很好的理由。需要支持 Windows 7 客户端吗？在这种情况下，UWP 不合适，但 WPF 合适。仍然可以以一种易于切换到其他技术的方式创建 WPF 应用程序，例如 UWP 和 Xamarin。

**注意：**
请阅读第 34 章，了解如何设计应用程序，以便在 WPF、UWP 和 Xamarin 之间共享尽可能多的代码。

如果创建 Web 应用程序，肯定应使用 ASP.NET Core 与 ASP.NET Core MVC。做这个选择时要排除 ASP.NET Web Forms。如果访问数据库，就应该使用 Entity Framework Core，应该选择 Managed Extensibility Framework 而不是 System.AddIn。

旧应用程序仍在使用 Windows Forms、ASP.NET Web Forms 和其他一些旧技术。只为改变现有的应用程序而使用新技术是没有意义的。进行修改必须有巨大的优势，例如，维护代码已经是一个噩梦，需要大量的重构以缩短客户要求的发布周期，或者使用一项新技术可以减少更新包的编码时间。根据旧有应用程序的类型，使用新技术可能不值得。可以允许应用程序仍使用旧技术，因为在未来的许多年仍将支持 Windows Forms 和 ASP.NET Web Forms。

本书的内容以新技术为基础，展示创建新应用程序的最佳技术。如果仍然需要维护旧应用程序，可以参考《C#高级编程(第 10 版)　C#6 & .ENT Core 1.0》，其中介绍了 ASP.NET Web Forms、WCF、Windows Forms、System.AddIn、Workflow Foundation 和其他仍然在.NET Framework 中可用的旧技术。

## 1.3 .NET 术语

什么是当前的.NET 技术？图 1-3 给出了.NET Framework、.NET Core 和 Mono 相互关联的总体情况。所有的.NET Framework 应用程序、.NET Core 应用程序和 Xamarin 应用程序如果是用.NET Standard 构建的，就都可以使用相同的库。这些技术共享相同的编译器平台、编程语言和运行库组件。它们不共享相同的运行库，但是它们在运行时共享组件。例如，.NET Framework 和.NET Core 使用即时(JIT)编译器 RyuJIT。

使用.NET Framework，可以创建 Windows Forms、WPF 和在 Windows 上运行的旧 ASP.NET 应用程序。

使用.NET Core，可以创建在不同平台上运行的 ASP.NET Core 和控制台应用程序。.NET Core 也由通用 Windows 平台(UWP)使用，但这并不能使 UWP 在 Linux 上可用。UWP 还利用了 Windows 运行库，它只能在 Windows 上使用。

Xamarin 提供了 Xamarin.IoS 和 Xamarin.Android 库，它们可以为 iPhone 和 Android 开发 C#应用程序。有了 Xamarin.Forms，就可以在两个移动平台之间共享用户界面。Xamarin 目前仍然基于 Mono 框架，Mono 框架是由 Xamarin 开发的.NET 变体。在某种程度上，这可能会变为.NET Core。然而，重要的是所有这些技术都可以使用为.NET 标准创建的相同的库。

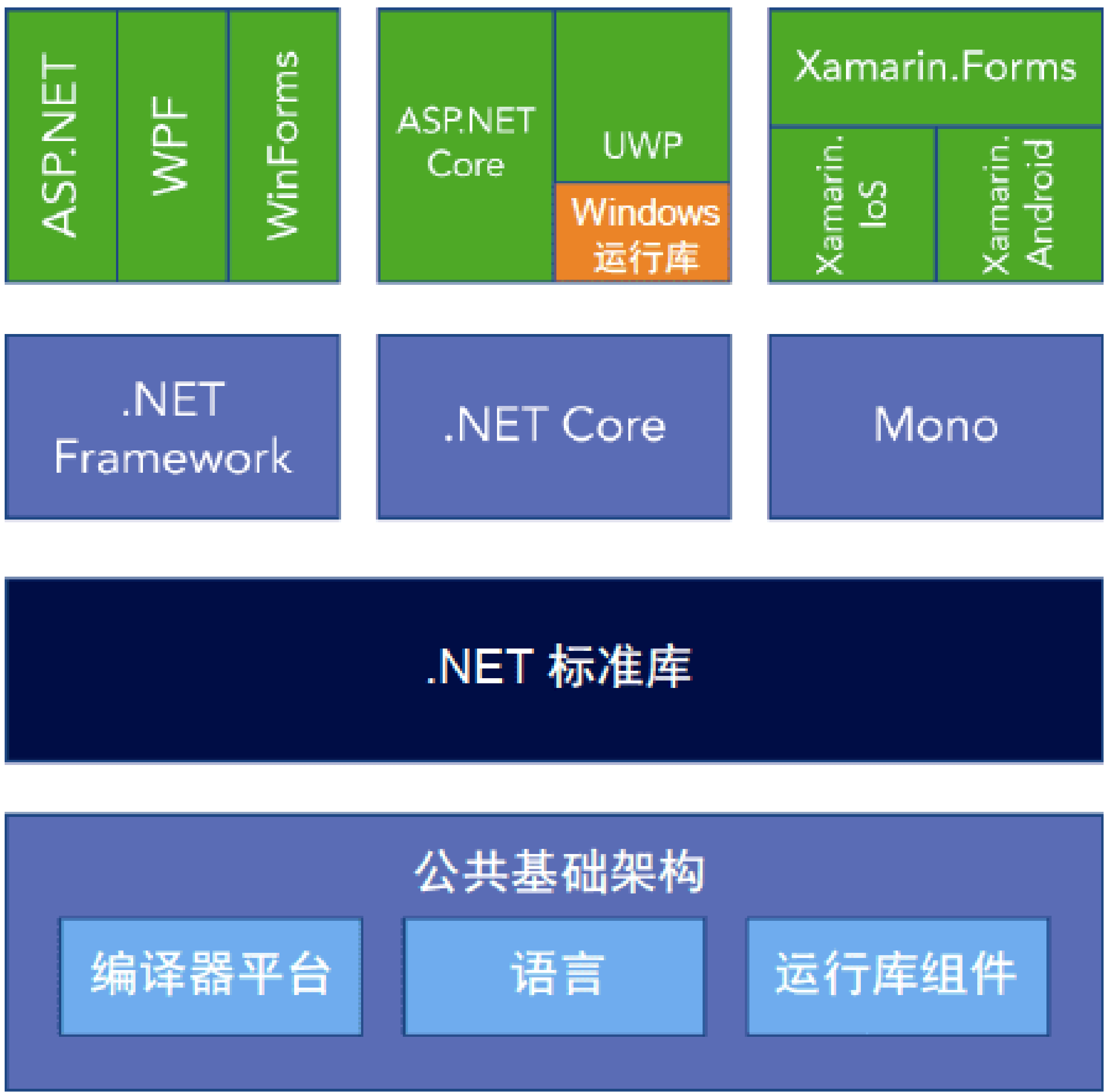



图 1-3

在图 1-3 的下半部分，可以看到.NET Framework、.NET Core 和 Mono 之间也有一些共享。运行库组件的代码是共享的，例如垃圾收集器和 RyuJIT(这是一个新的 JIT 编译器，可以将 IL 代码编译为本地代码)。垃圾收集器由 CLR、CoreCLR 和.NET Native 使用。CLR 和 CoreCLR 使用 RyuJIT 即时编译器。所有这些平台都使用.NET Compiler Platform(也称为 Roslyn)和编程语言。

### 1.3.1 .NET Framework

.NET Framework 4.7 是.NET Framework 在过去 15 年不断增强的结果。1.2 节讨论的许多技术都基于这个框架。这个框架用于创建 Windows Forms 和 WPF 应用程序。.NET Framework 4.7 还提供了 Windows Forms 的增强功能，比如对 High DPI 的支持。

如果希望继续使用 ASP.NET Web Forms，就应选择 ASP.NET 4.7 和.NET Framework 4.7。否则，就需要重写一些代码，以移动到.NET Core。根据源代码的质量和添加新特性的需要，重写代码可能是值得的。

### 1.3.2 .NET Core

.NET Core 是新的.NET，所有新技术都使用它，是本书的一大关注点。这个框架是开源的，可以在 http://www.github.com/dotnet 上找到它。运行库是 CoreCLR 库；包含集合类的框架、文件系统访问、控制台和 XML 等都在 CoreFX 库中。

.NET Framework 要求必须在系统上安装应用程序需要的特定版本，而在.NET Core 1.0 中，框架(包括运行库)是与应用程序一起交付的。以前，把 ASP.NET Web 应用程序部署到共享服务器上有时可能有问题，因为提供程序安装了旧版本的.NET。这种情况已经一去不复返了。现在可以同时提交应用程序和运行库，而不依赖服务器上安装的版本。

.NET Core 以模块化的方式设计。该框架分成数量很多的 NuGet 包。这样就不必处理所有的包了，而可以用元包来引用一起工作的小包。使用.NET Core 2.0 和 ASP.NET Core 2.0，甚至可以改进元包。通过 ASP.NET Core 2.0，只需要引用 Microsoft.AspNetCore.All，就可以得到 ASP.NET Core web 应用程序通常需要的所有包。

.NET Core 可以很快更新。即使更新运行库，也不影响现有的应用程序，因为运行库与应用程序一起安装。现在，微软公司可以增强.NET Core，包括运行库，发布周期更短。

**注意：**
为了使用.NET Core 开发应用程序，微软公司创建了新的命令行实用程序.NET Core Command Line (CLI)。这些工具参见本章后面的内容。

### 1.3.3 .NET Standard

.NET Standard 不是一个实现，而是一个协定。本协定规定了需要实现哪些 API。.NET Framework、.NET Core 和 Xamarin 实现了这个标准。

标准是有版本的。在每个版本中都添加了额外的 API。根据需要的 API，可以选择库的标准版本。需要检查所选平台是否符合所需版本的标准。

可以在 https://docs.microsoft.com/en-us/dotnet/standard/net-standard 中找到.NET Standard 的平台支持表。以下是需要了解的最重要的部分：

- .NET Core 1.1 支持.NET Standard 1.6，.NET Core 2.0 支持.NET Standard 2.0
- .NET Framework 4.6.1 支持.NET Standard 2.0。
- UWP 构建了 16299，后来支持.NET Standard 2.0；旧版本只支持.NET Standard 1.4。
- 通过 Xamarin 使用.NET Standard 2.0，需要 Xamarin.iOS 10.14 和 Xamarin.Android 8.0。

**注意：**
请阅读第 19 章中关于.NET Standard 的详细信息。

### 1.3.4 NuGet 包

在早期，程序集是应用程序的可重用单元。添加对程序集的一个引用，以使用自己代码中的公共类型和方法，此时，仍可以这样使用(一些程序集必须这样使用)。然而，使用库可能不仅意味着添加一个引用并使用它。使用库也意味着一些配置更改，或者可以通过脚本来利用一些特性。这是在 NuGet 包中打包程序集的一个原因。

NuGet 包是一个 zip 文件，其中包含程序集(或多个程序集)、配置信息和 PowerShell 脚本。

使用 NuGet 包的另一个原因是，它们很容易找到，它们不仅可以从微软公司找到，也可以从第三方找到。NuGet 包很容易在 NuGet 服务器 http://www.nuget.org 上获得。

在 Visual Studio 项目的引用中，可以打开 NuGet 包管理器(NuGet Package Manager，见图 1-4)，在该管理器中可以搜索包，并将其添加到应用程序中。这个工具允许搜索还没有发布的包(包括预发布选项)，定义应该在哪个 NuGet 服务器中搜索包。搜索包的一个地方是自己的共享目录，其中放置了内部使用的包。

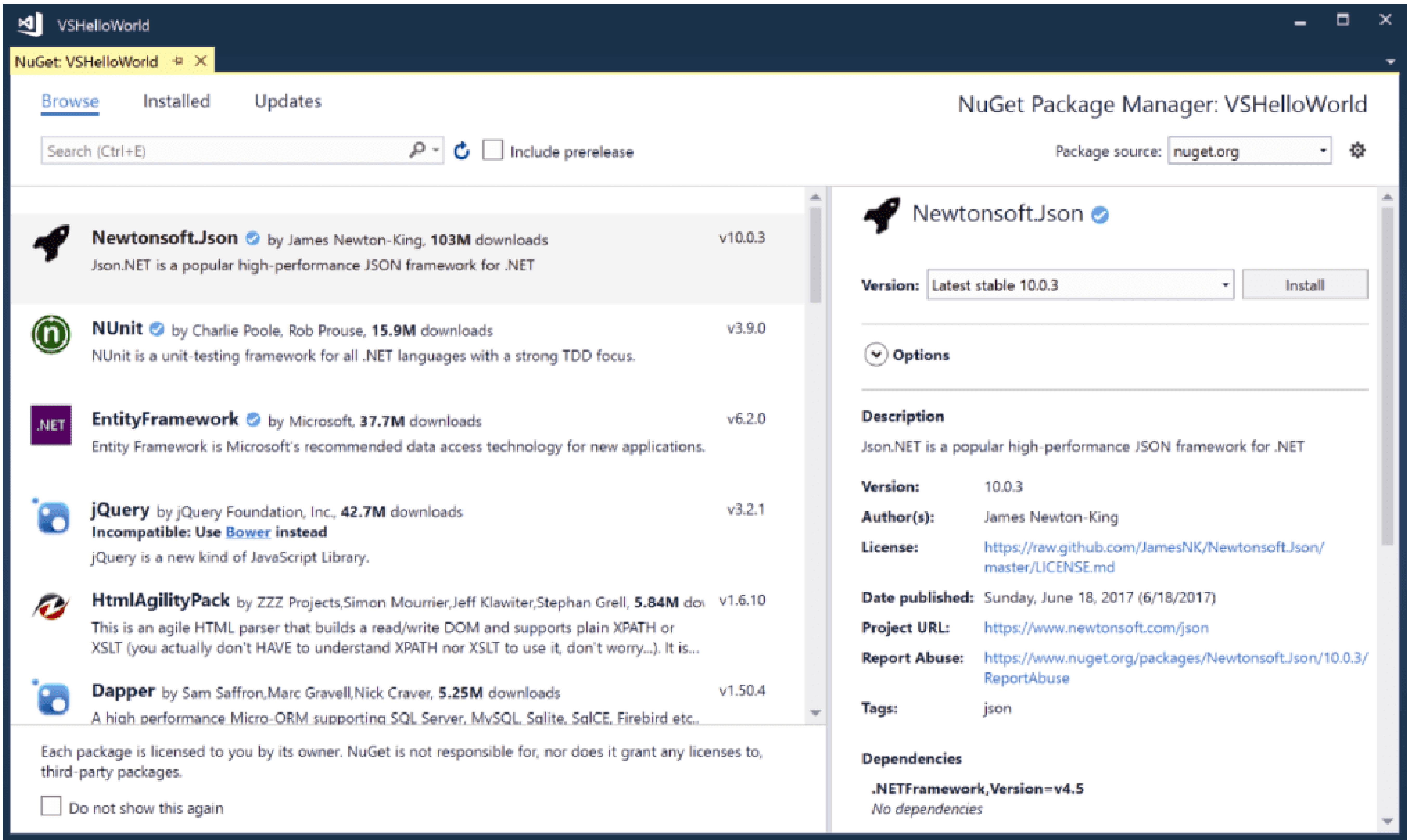



图 1-4

> **注意：**
> 使用 NuGet 服务器中的第三方包时，如果一个包以后才能使用，就总是有风险。还需要检查包的支持可用性。使用包之前，总要检查项目的链接信息。对于包的来源，可以选择 Microsoft and .NET，只获得微软公司支持的包。第三方包也包括在 Microsoft and .NET 部分中，但它们是微软公司支持的第三方包。

> **注意：**
> NuGet 包管理器的更多信息参见第 18 章。

### 1.3.5 名称空间

可用于.NET 的类组织在名称以 System 开头的名称空间中。表 1-3 描述的名称空间提供了层次结构的思路。

表 1-3

| 名 称 空 间 | 说 明 |
|---|---|
| System.Collections | 这是集合的根名称空间。子名称空间也包含集合，如 System.Collections.Concurrent 和 System.Collections. Generic |
| System.Data | 这是访问数据库的名称空间。System.Data.SqlClient 包含访问 SQL Server 的类 |
| System.Diagnostics | 这是诊断信息的根名称空间，如事件记录和跟踪(在 System.Diagnostics.Tracing 名称空间中) |
| System.Globalization | 该名称空间包含的类用于全球化和本地化应用程序 |
| System.IO | 这是文件 IO 的名称空间，其中的类访问文件和目录，包括读取器、写入器和流 |
| System.Net | 这是核心网络的名称空间，比如访问 DNS 服务器，用 System.Net.Sockets 创建套接字 |
| System.Threading | 这是线程和任务的根名称空间。任务在 System.Threading.Tasks 中定义 |

**注意：**

一些新的.NET 类使用名称以 Microsoft 开头而不是以 System 开头的名称空间，比如用于 Entity Framework Core 的 Microsoft.EntityFrameworkCore，用于新的依赖关系注入框架的 Microsoft.Extensions.DependencyInjection。

### 1.3.6 公共语言运行库

UWP 利用 Native .NET 通过 AOT Compiler 把 IL 编译成本地代码。这与 Xamarin.iOS 类似。在所有其他场景中，使用.NET Framework 的应用程序和使用.NET Core 1.0 的应用程序都需要 CLR(Common Language Runtime，公共语言运行库)。然而，.NET Core 使用 CoreCLR，而.NET Framework 使用 CLR。那么，CLR 的作用是什么？

在 CLR 执行应用程序之前，编写好的源代码(使用 C#或其他语言编写的代码)都需要编译。在.NET 中，编译分为两个阶段：

(1) 将源代码编译为 Microsoft 中间语言(Intermediate Language，IL)。

(2) CLR 把 IL 编译为平台专用的本地代码。

IL 代码在.NET 程序集中可用。在运行时，JIT 编译器编译 IL 代码，创建特定于平台的本地代码。

新的 CLR 和 CoreCLR 包括一个新的 JIT 编译器 RyuJIT。新的 JIT 编译器不仅比以前的版本快，还在用 Visual Studio 调试时更好地支持 Edit & Continue 特性。Edit & Continue 特性允许在调试时编辑代码，可以继续调试会话，而不需要停止并重新启动过程。

CLR 还包括一个带有类型加载器的类型系统，类型加载器负责从程序集中加载类型。类型系统中的安全基础设施验证是否允许使用某些类型系统结构，如继承。

创建类型的实例后，实例还需要销毁，内存也需要回收。CLR 的另一个功能是垃圾收集器。垃圾收集器从托管堆中清除不再引用的内存。

CLR 还负责线程的处理。在 C#中创建托管的线程不一定来自底层操作系统。线程的虚拟化和管理由 CLR 负责。

**注意：**

如何在 C#中创建和管理线程参见第 21 章和 22 章。第 17 章介绍了垃圾收集器和清理内存的方法。

### 1.3.7 Windows 运行库

从 Windows 8 开始，Windows 操作系统提供了另一种框架：Windows 运行库(Windows Runtime)。这个运行库由 WUP(Windows Universal Platform，Windows 通用平台)使用，Windows 8 使用第 1 版，Windows 8.1 使用第 2 版，Windows 10 使用第 3 版。

与.NET Framework 不同，这个框架是使用本地代码创建的。当它用于.NET 应用程序时，所包含的类型

和.NET 类似。在语言投射的帮助下，Windows 运行库可以用于 JavaScript、C++和.NET 语言，它看起来像编程环境的本地代码。不仅方法因区分大小写而行为不同，方法和类型也可以根据所处的位置有不同的名称。

Windows 运行库提供了一个对象层次结构，它在以 Windows 开头的名称空间中组织。这些类没有复制.NET Framework 的很多功能；相反，提供了额外的功能，用于在 UWP 上运行的应用程序。如表 1-4 所示。

表 1-4

| 名 称 空 间 | 说　明 |
|---|---|
| Windows.ApplicationModel | 这个名称空间及其子名称空间(如 Windows.ApplicationModel.Contracts)定义了类，用于管理应用程序的生命周期，与其他应用程序通信 |
| Windows.Data | Windows.Data 定义了子名称空间，来处理文本、JSON、PDF 和 XML 数据 |
| Windows.Devices | 地理位置、智能卡、服务设备点、打印机、扫描仪等设备可以用 Windows.Devices 子名称空间访问 |
| Windows.Foundation | Windows.Foundation 定义了核心功能。集合的接口用名称空间 Windows.Foundation.Collections 定义。这里没有具体的集合类。相反，.NET 集合类型的接口映射到 Windows 运行库类型 |
| Windows.Media | Windows.Media 是播放、捕获视频和音频、访问播放列表和语音输出的根名称空间 |
| Windows.Networking | 这是套接字编程、数据后台传输和推送通知的根名称空间 |
| Windows.Security | Windows.Security.Credentials 中的类提供了密码的安全存储区，Windows.Security.Credentials.UI 提供了一个选择器，用于从用户处获得凭据 |
| Windows.Services.Maps | 这个名称空间包含用于定位服务和路由的类 |
| Windows.Storage | 有了 Windows.Storage 及其子名称空间，就可以访问文件和目录，使用流和压缩 |
| Windows.System | Windows.System 名称空间及其子名称空间提供了系统和用户的信息，也提供了一个启动其他应用程序的启动器 |
| Windows.UI.Xaml | 在这个名称空间中，可以找到很多用于用户界面的类型 |

## 1.4 用.NET Core CLI 编译

在本书的许多章节中并不需要 Visual Studio，而可以使用任何编辑器和命令行。要创建和编译应用程序，可以使用.NET Core 命令行接口(Command Line Interface，CLI)。下面看看如何设置系统，以及如何使用这个工具。

### 1.4.1 设置环境

安装了 Visual Studio 2017 和最新的更新包后，就可以立即启动 CLI 工具。可以在没有 Visual Studio 2017 的情况下建立一个系统，还可以在 Linux 和 OS X 上使用大部分示例。为了下载环境的应用程序，只需要访问 https://dot.net 并单击 Get Started 按钮。从这里，可以下载 Windows、Linux 和 macOS 的.NET SDK。

对于 Windows，可以下载安装 SDK 的可执行文件。使用 Linux，需要选择 Linux 发行版来获得相应的命令：

- 通过 Red Hat 和 CentOS，使用 yum 安装.NET SDK
- 通过 Ubuntu 和 Debian，使用 apt-get
- 通过 Fedora，使用 dnf install
- 通过 SLES/openSUSE，使用 zipper install
- 要在 Mac 上安装.NET SDK，可以下载.pkg 文件。

在 Windows 上，不同版本的.NET Core 运行库以及 NuGet 包安装在用户配置文件中。使用.NET 时，这个文件夹的大小会增加。随着时间的推移，会创建多个项目，NuGet 包不再存储在项目中，而是存储在这个用户专用的文件夹中。这样做的优势在于，不需要为每个不同的项目下载 NuGet 包。这个 NuGet 包下载后，它就在

系统上。因为不同版本的 NuGet 包和运行库都是可用的，所有的不同版本都存储在这个文件夹中。不时地检查这个文件夹，删除不再需要的旧版本，可能很有趣。

安装.NET Core CLI 工具，要把 dotnet 工具作为入口点来启动所有这些工具。只需要启动：

```
> dotnet  --help
```

会看到，dotnet 工具的所有不同选项都可用。许多选项都有简化符号。要获得帮助，可以输入：

```
> dotnet -h
```

### 1.4.2　创建应用程序

dotnet 工具提供了一种简单的方法创建“Hello World!”应用程序。输入如下命令：

```
> dotnet new console --output HelloWorld
```

这个命令创建一个新的 HelloWorld 目录并添加源代码文件 Program.cs 和项目文件 HelloWorld.csproj。从.NET Core 2.0 开始，这个命令还包括 dotnet restore，所有的 NuGet 包都会下载。要查看应用程序使用的库的依赖项和版本列表，可以检查 obj 子目录中的文件 project.assets. json。如果不使用选项--output (或-o 速记符号)，文件就在当前目录中生成。

生成的源代码如下所示(代码文件 HelloWorld/Program.cs)：

```
using System;

namespace HelloWorld
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("Hello World!");
    }
  }
}
```

自从 20 世纪 70 年代 Brian Kernighan 和 Dennis Ritchie 撰写了《C 编程语言》一书，使用“Hello World”应用程序开始，学习编程语言就变成一种传统。使用.NET Core CLI，这个程序会自动生成。

下面看看这个程序的语法。Main()方法是.NET 应用程序的入口点。CLR 在启动时调用静态 Main()方法。Main()方法需要放到一个类中。这里，这个类命名为 Program，但是可以给它指定任何名称。

Console.WriteLine 调用 Console 类的 WriteLine()方法。Console 类在 System 名称空间中。不需要编写 System.Console.WriteLine 调用该方法；System 名称空间在源文件的顶部使用 using 声明打开。

在编写源代码之后，需要编译代码来运行它。

创建的项目配置文件名为 HelloWorld.csproj。与较老版本的 csproj 文件相比，新项目文件减少为几行，有几个默认值：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>
</Project>
```

对于项目文件，OutputType 定义了输出的类型。对于控制台应用程序，该类型是 Exe。TargetFramework 指定了用于构建应用程序的框架和版本。在样例项目中，应用程序是使用.NET Core 2.0 构建的。可以将此元素更改为 TargetFramework，并指定多个框架，如 netcoreapp2.0。net47 用于为.NET Framework 4.7 和.NET Core 2.0 构建应用程序(项目文件 HelloWorld/HelloWorld.csproj)：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFrameworks>netcoreapp2.0;net47</TargetFrameworks>
  </PropertyGroup>
</Project>
```

sdk 属性指定项目使用的 SDK。微软有两个主要的 SDK：Microsoft.NET.Sdk 用于控制台应用程序，Microsoft.NET.Sdk.Web 用于 ASP.NET Core web 应用程序。

不需要向项目添加源文件。在编译时，会自动添加同一目录和子目录下扩展名为 cs 的文件。扩展名为 resx 的资源文件是自动添加的，用于嵌入资源。可以更改默认行为，并显式排除/包含文件。

也不需要添加.NET Core 包。通过指定目标框架 netcoreapp2.0，引用许多其他包的元包 Microsoft.NetCore.App 会自动包含在内。

### 1.4.3　构建应用程序

要构建应用程序，需要将当前目录更改为应用程序的目录，并启动 dotnet build。为.NET Core 2.0 和.NET Framework 4.7 编译时，输出如下：

```
> dotnet build
Microsoft (R) Build Engine version 15.5.179.9764 for .NET Core
Copyright (C) Microsoft Corporation. All rights reserved.

  Restore completed in 19.8 ms for
    C:\procsharp\Intro\HelloWorld\HelloWorld.csproj.
  HelloWorld -> C:\procsharp\Intro\HelloWorld\bin\Debug\net47\HelloWorld.exe
  HelloWorld ->
    C:\procsharp\Intro\HelloWorld\bin\Debug\netcoreapp2.0\HelloWorld.dll

Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:01.58
```

**注意：**
命令 dotnet new 和 dotnet build 现在包括恢复 NuGet 包。使用 dotnet restore 还可以显式地恢复 NuGet 包。

编译过程的结果是在 bin/debug/[netcoreapp2.0|net47]文件夹中的程序集包含 Program 类的 IL 代码。如果比较.NET Core 与.NET 4.7 的构建结果，会发现一个包含了 IL 代码和.NET Core 的 DLL，以及一个包含了 IL 代码和.NET 4.7 的 EXE。为.NET Core 生成的程序集有一个对 System.Console 程序集的依赖项，而.NET 4.6 程序集在 mscorlib 程序集中找到 Console 类。

要构建发布代码，就需要指定选项--configuration Release (简写为-c Release)：

```
> dotnet build --configuration Release
```

以下章节中的一些代码示例使用了 C# 7.1 或 C# 7.2 提供的功能。默认情况下，使用编译器的最新主版本，即 C# 7.0。要启用新版本的 C#，需要在项目文件中指定这一点，如下面的项目文件部分所示。这里，配置了 C#编译器的最新版本。

```
<PropertyGroup>
  <LangVersion>latest</LangVersion>
</PropertyGroup>
```

### 1.4.4　运行应用程序

要运行应用程序，可以使用 dotnet run 命令。

```
> dotnet run
```

如果项目文件面向多个框架，就需要通过--framework 选项告诉 dotnet run 命令，使用哪个框架来运行应用程序。这个框架必须通过 csproj 文件来配置。使用样例应用程序，可以在恢复信息之后得到如下输出：

```
> dotnet run --framework netcooreapp2.0
Microsoft (R) Build Engine version 15.5.179.9764 for .NET Core
Copyright (C) Microsoft Corporation. All rights reserved.

  Restore completed in 20.65 ms for
    C:\procsharp\Intro\HelloWorld\HelloWorld.csproj.
```

```
Hello World!
```

在生产系统中，不使用 dotnet run 运行应用程序，而可以使用 dotnet 和库的名称：

```
> dotnet bin/debug/netcoreapp2.0/HelloWorld.dll
```

还可以创建可执行文件，但可执行文件是特定于平台的。

**注意：**

前面是在 Windows 上构建和运行“Hello World!”应用程序，而 dotnet 工具在 Linux 和 OS X 上的工作方式是相同的。可以在这两个平台上使用相同的 dotnet 命令。

本书的重点是 Windows，因为 Visual Studio 2017 提供了一个比其他平台更强大的开发平台，但本书的许多代码示例是基于.NET Core 的，所以也能够在其他平台上运行。还可以使用 Visual Studio Code(一个免费的开发环境)，直接在 Linux 和 OS X 上开发应用程序，参见 1.7 节，了解 Visual Studio 不同版本的更多信息。

### 1.4.5 创建 Web 应用程序

还可以使用.NET Core CLI 创建 Web 应用程序。启动 dotnet new 时，可以看到可用的模板列表(参见图 1-5)。

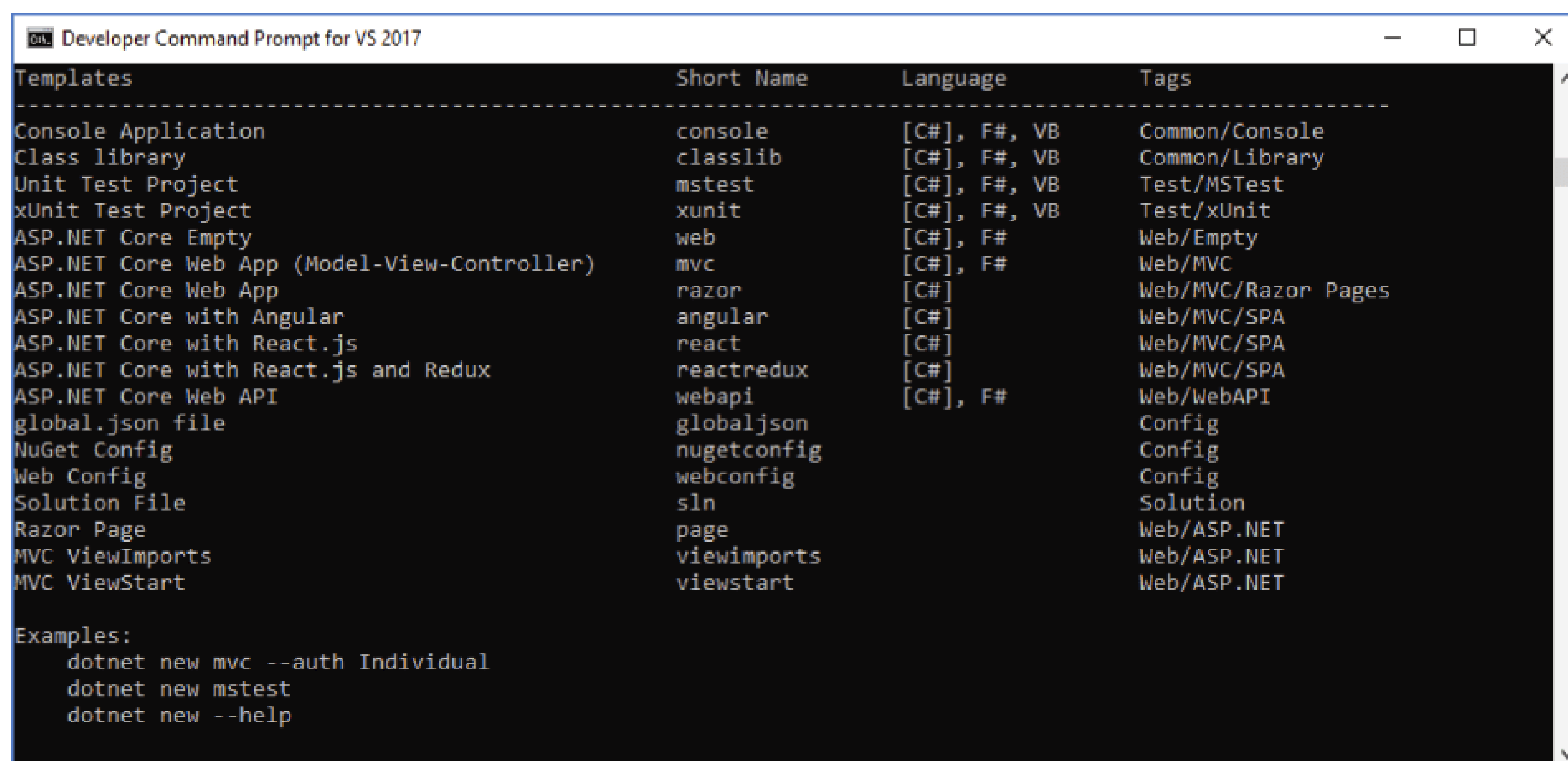



图 1-5

下面的命令

```
> dotnet new mvc -o WebApp
```

会使用 ASP.NET Core MVC 创建新的 ASP.NET Core web 应用程序。切换到 WebApp 文件夹之后，使用下述命令构建和运行程序：

```
> dotnet build
> dotnet run
```

运行上述代码会启动 ASP.NET Core 的 Kestrel 服务器，来监听端口 5000。可以打开浏览器访问从这个服务器返回的页面，如图 1-6 所示。

### 1.4.6 发布应用程序

使用 dotnet 工具，可以创建一个 NuGet 包并发布应用程序来进行部署。首先创建应用程序的框架依赖部署。这减少了发布所需的文件。

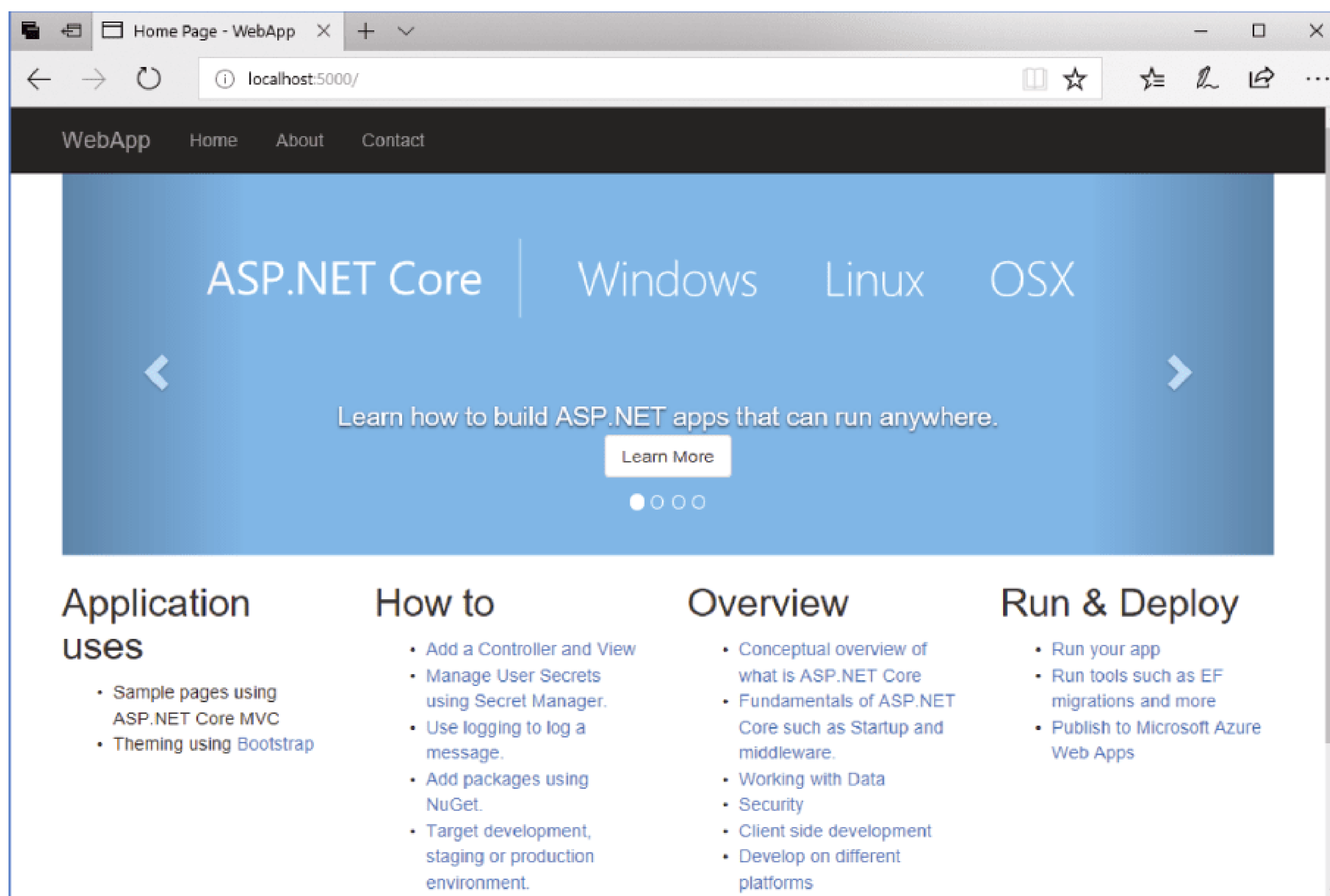



图 1-6

使用之前创建的控制台应用程序，只需要以下命令来创建发布所需的文件。使用-f 选择框架，使用-c 选择版本配置。

```
> dotnet publish -f netcoreapp2.0 -c Release
```

发布所需的文件放在 bin/Release/netcoreapp2.0/publish 目录中。

在目标系统上使用这些文件进行发布，也需要运行库。在 https://www.microsoft.com/net/download/上可以找到运行库的下载和安装说明。

在.NET Framework 中，相同的安装运行库可以由不同的.NET Framework 版本使用(例如，.NET Framework 4.0 运行库和更新包可以在.NET Framework 4.7、4.6、4.5、4.0 等应用程序中使用)，与.NET Framework 相反，对于.NET Core，要运行应用程序，就需要相同的运行库版本。

**注意：**

*如果应用程序使用了额外的 NuGet 包，这些就需要在 csproj 文件中引用，并且库需要与应用程序一起交付。阅读第 19 章，了解更多信息。*

**自包含部署**

应用程序不需要在目标系统上安装运行库，而是可以用它交付运行库。这就是所谓的自包含部署。

平台不同，运行库就不同。因此，对于自包含部署，需要通过在项目文件中指定 RuntimeIdentifiers，来指定支持的平台，如下面的项目文件所示。这里，指定了 Windows 10、macOS 和 Ubuntu Linux 的运行库标识符(项目文件 SelfContainedHelloWorld/SelfContainedHelloWorld.csproj)：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>
  <PropertyGroup>
    <RuntimeIdentifiers>
      win10-x64;ubuntu-x64;osx.10.11-x64;
    </RuntimeIdentifiers>
  </PropertyGroup>
</Project>
```

**注意：**

在 https://docs.microsoft.com/en-us/dotnet/core/rid-catalog 的.NET Core Runtime Identifier(RID)类别中可以获取不同平台和版本的所有运行库标识符。

现在可以为所有不同的平台创建发布文件：

```
> dotnet publish -c Release -r win10-x64
> dotnet publish -c Release -r osx.10.11-x64
> dotnet publish -c Release -r ubuntu-x64
```

在运行这些命令之后，可以在 Release/[win10- x64|osx.10.11-x64|ubuntu-x64]/publish 目录中找到发布所需要的文件。随着.NET Core 2.0 的规模越来越大，发布的规模也越来越大。在这些目录中，可以找到特定于平台的可执行文件，可以在不使用 dotnet 命令的情况下直接启动它。

## 1.5 使用 Visual Studio 2017

接下来使用 Visual Studio 2017 代替命令行。本节将介绍 Visual Studio 中最重要的部分来开始工作。Visual Studio 的更多特性在第 18 章中介绍。

### 1. 安装 Visual Studio 2017

Visual Studio 2017 提供了一个新的安装程序，它可以更容易安装需要的产品。使用安装程序，可以选择开发应用程序所需的工作负载(参见图 1-7)。为了涵盖本书的所有章节，安装这些工作负载：

- UWP 开发
- .NET Desktop 开发
- ASP.NET 和 Web 开发
- Azure 开发
- 移动开发与.NET
- .NET Core 跨平台开发

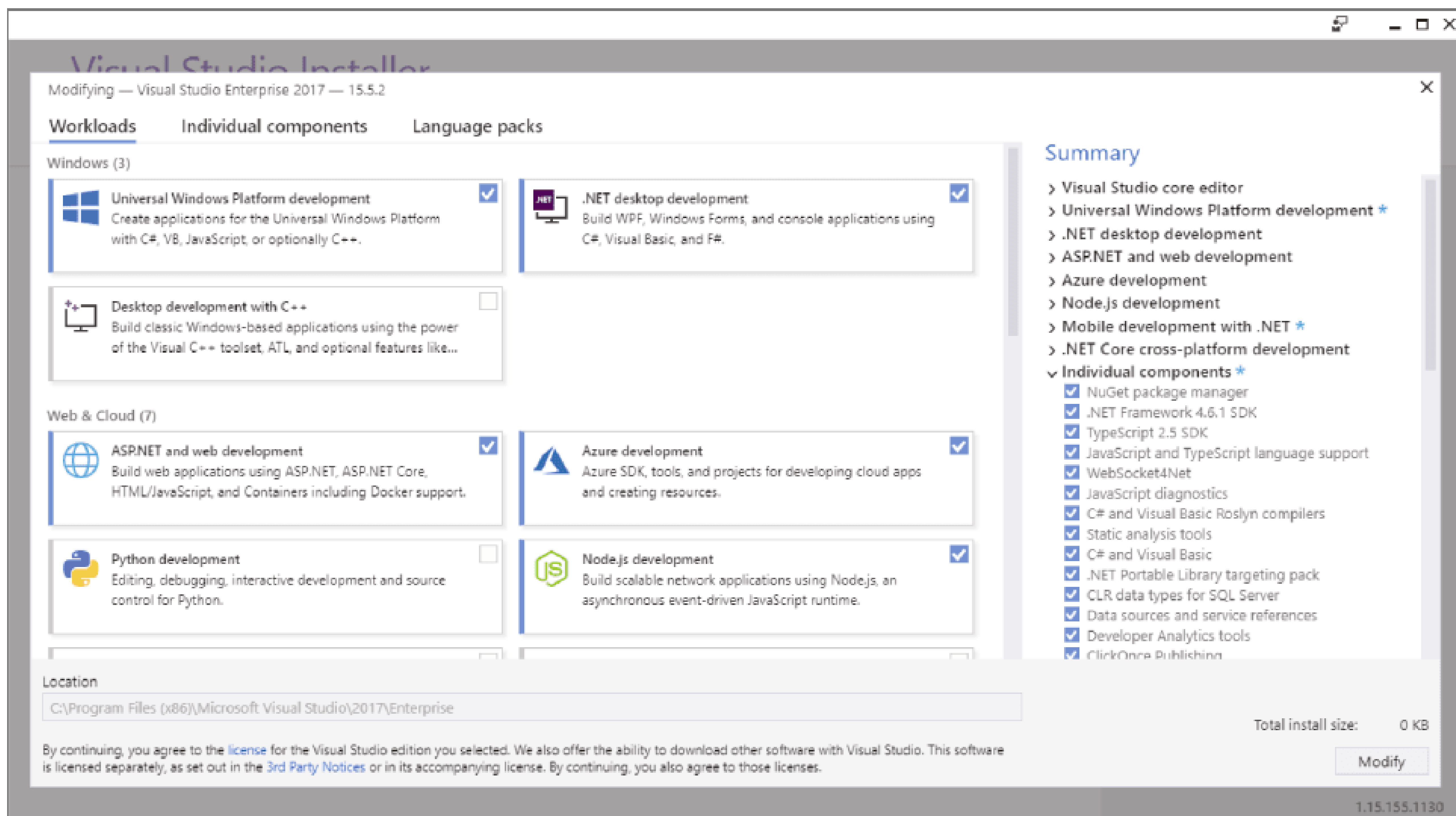



图 1-7

### 2. 创建一个项目

你可能会被大量的菜单项和 Visual Studio 中的许多选项所淹没。要在本书的第 1 章中创建简单的应用程序，

只需要使用 Visual Studio 的一小部分特性。同样，这本完整的参考书只涵盖了可以用 Visual Studio 完成的一部分操作。Visual Studio 的许多特性是为遗留应用程序以及其他编程语言提供的。

在启动 Visual Studio 之后，首先要创建一个新项目。选择菜单 File | New | Project。打开如图 1-8 所示的对话框，其中列出了可以用来创建新项目的项目项的列表。

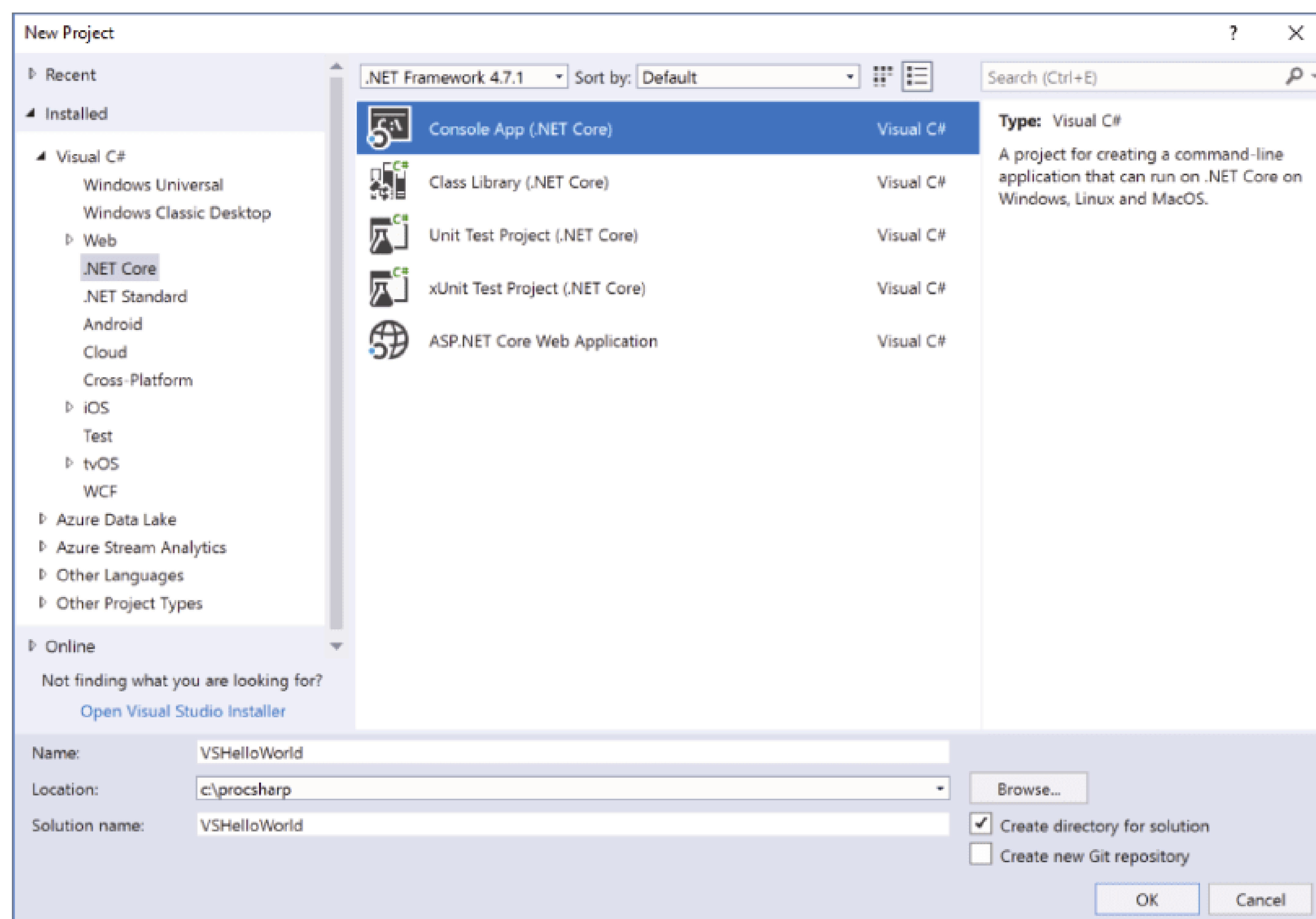



图 1-8

本书主要讨论的是 Visual C#项目项的一个子集。在本章中，选择.NET Core 类别和项目模板 Console App (.NET Core)。在如图 1-8 所示的对话框顶部，选择了.NET Framework 版本。不要混淆，这个选择并不适用于.NET Core 项目。

在此对话框的下半部分，可以输入应用程序的名称，选择存储项目的文件夹，并为解决方案输入一个名称。解决方案可以包含多个项目。

单击 OK 按钮，创建“Hello World!”应用程序。

### 3. 使用 Solution Explorer

在 Solution Explorer 中(参见图 1-9)，可以看到解决方案、属于解决方案的项目以及项目中的文件。可以选择能进入类和类成员的源代码文件。

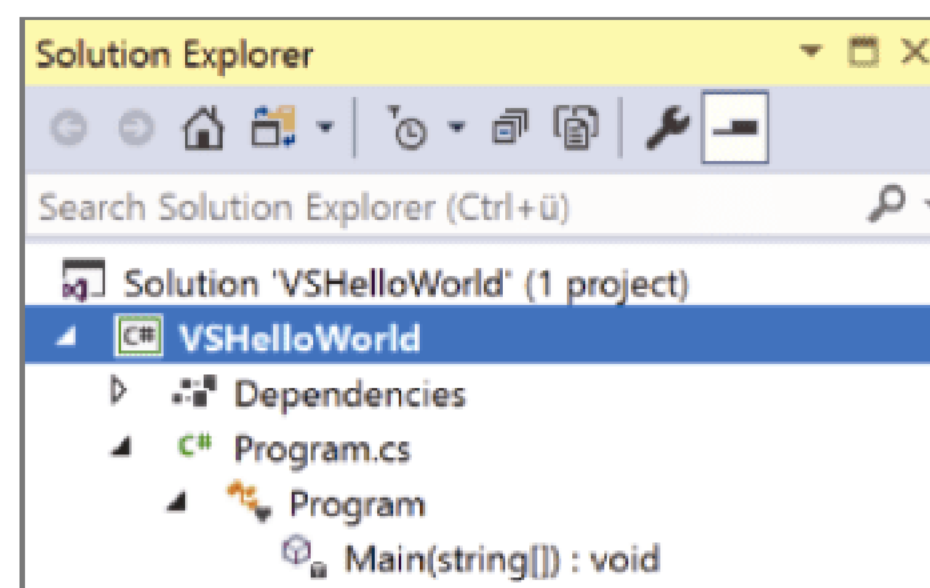



图 1-9

在 Solution Explorer 中选择一项，并单击鼠标右键或按下键盘上的应用程序键时，会打开该项的上下文菜单，如图 1-10 所示。可用的菜单取决于选择的项，以及与 Visual Studio 一起安装的特性。

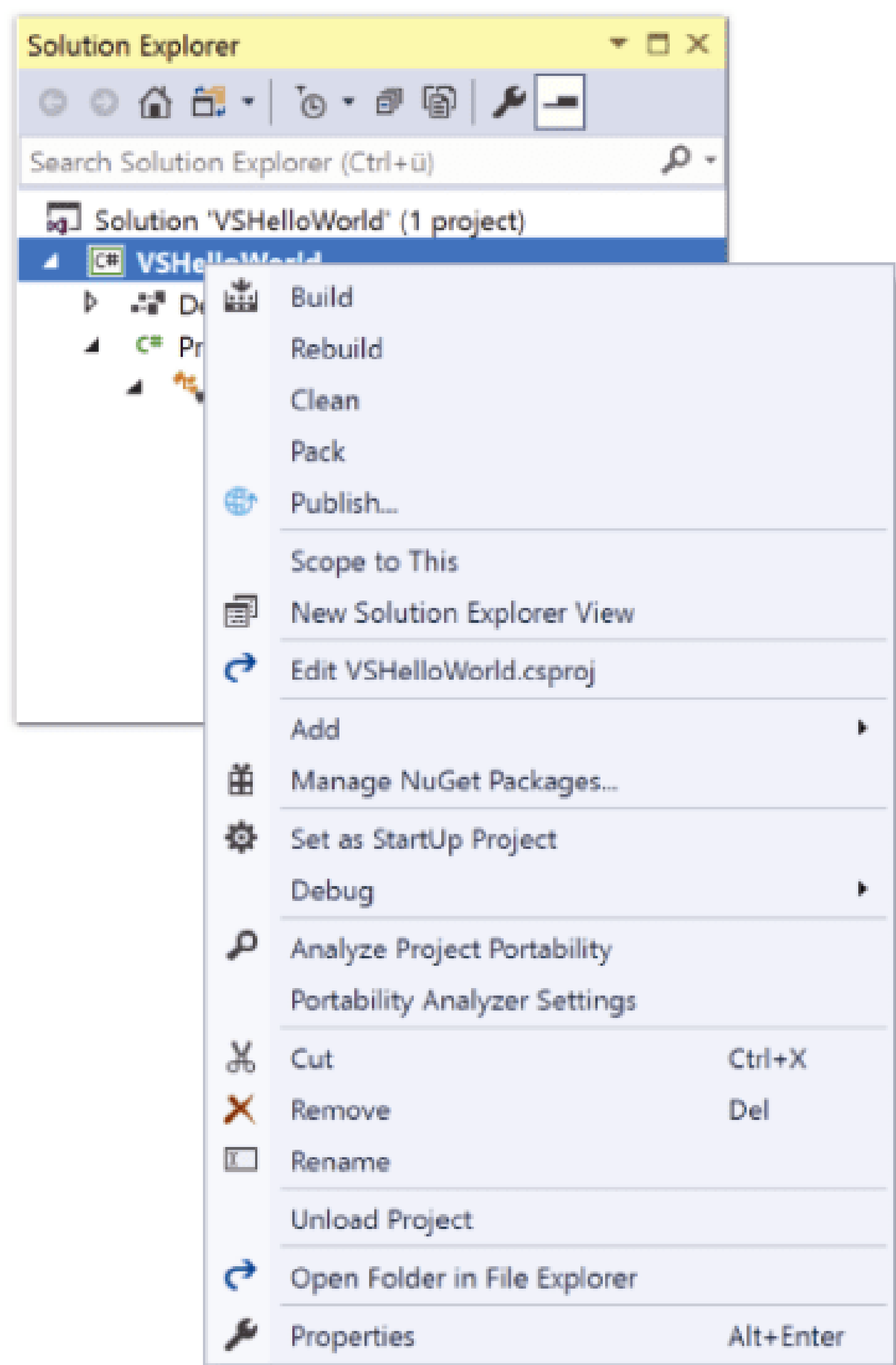



图 1-10

打开项目的上下文菜单时，其中有一个菜单项是用于编辑项目文件的。此选项会打开项目文件 VSHelloWorld.csproj，其内容与使用.NET Core CLI 时看到的内容相同。

### 4. 配置项目属性

为了配置项目属性，应在 Solution Explorer 中单击项目的上下文菜单，再单击 Properties，或选择 Project | VSHelloWorld Properties。打开如图 1-11 所示的视图。在这里，可以配置项目的不同设置，如要使用的.NET Core 版本(假定已经安装了多个框架)、构建设置、在构建过程中应该调用的命令、包配置以及在调试应用程序时使用的参数和环境变量。如前所述，对于一些代码示例，C# 7.0 是不够的。可以使用 Build 类别配置 C#编译器的不同版本。单击 Advanced 按钮将打开 Advanced Build Settings 对话框(参见图 1-12)。在这里，可以配置 C#编译器的版本。这个选择会进入 csproj 项目配置文件。

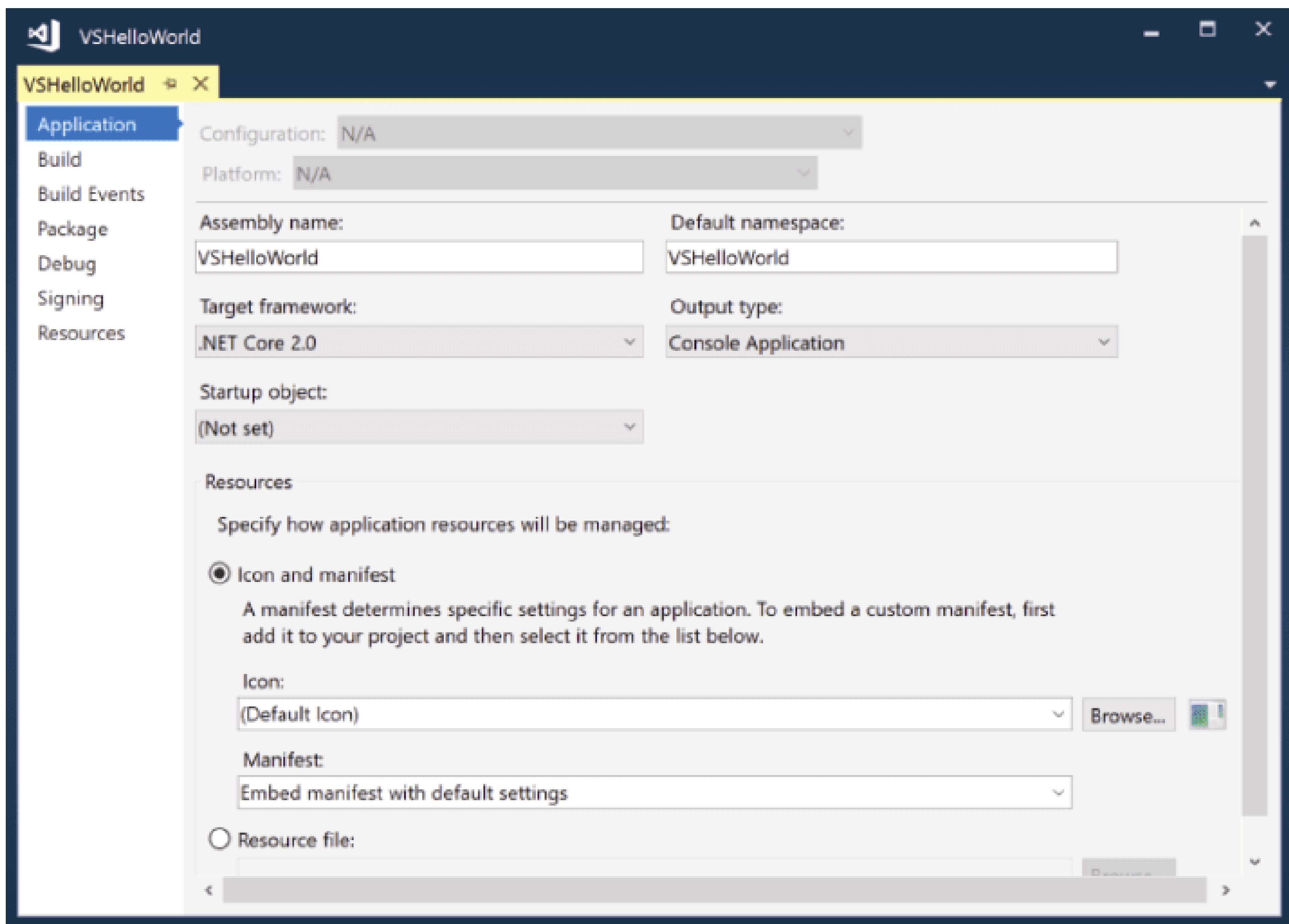



图 1-11

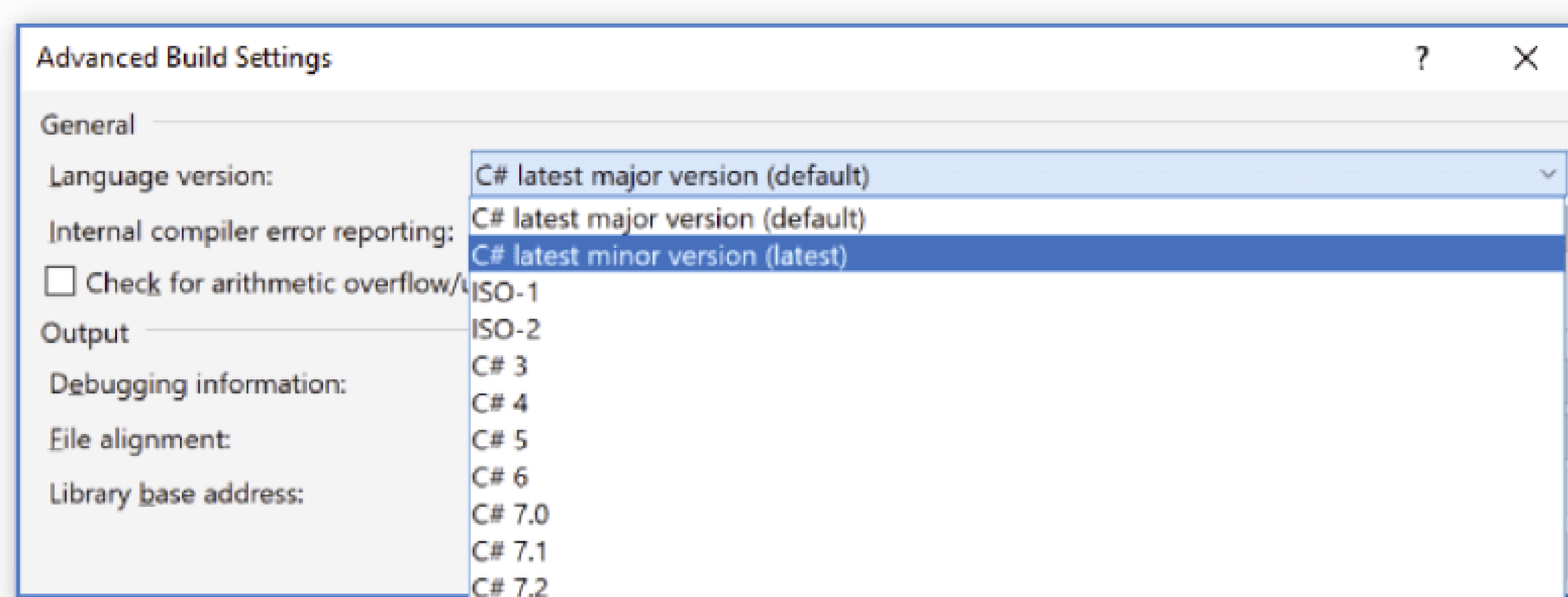



图 1-12

**注意：**

*在对项目属性进行更改时，需要确保在对话框顶部选择正确的配置。如果只使用 Debug 配置更改 C#编译器的版本，那么当使用新的 C#语言特性时，构建版本代码就会失败。对于想要的所有配置设置，请选择配置 AllConfigurations。*

### 5. 了解编辑器

Visual Studio 编辑器非常强大。它提供了智能感知功能，该功能可以在按下 Tab 键时，调用方法和属性，并完成输入。在键入时进行编译，可以立即看到带有下划线代码的语法错误。将鼠标指针悬停在下划线的文本上，会弹出一个包含错误描述的小框。

代码编辑器的一个重要的高效特性是代码片段。它们会减少需要输入的内容。只要在编辑器中输入 cw，再按 Tab 键，编辑器就会创建 Console.WriteLine();。Visual Studio 附带了许多代码片段，选择 Tools | Code Snippets Manager，打开 Code Snippets Manager 对话框(参见图 1-13)，可以看到这些代码片段。在这里，可以在 Language 字段中给用 C#语言定义的代码片段选择 CSharp，选择组 Visual C#可以查看为 C#预定义的所有代码片段。

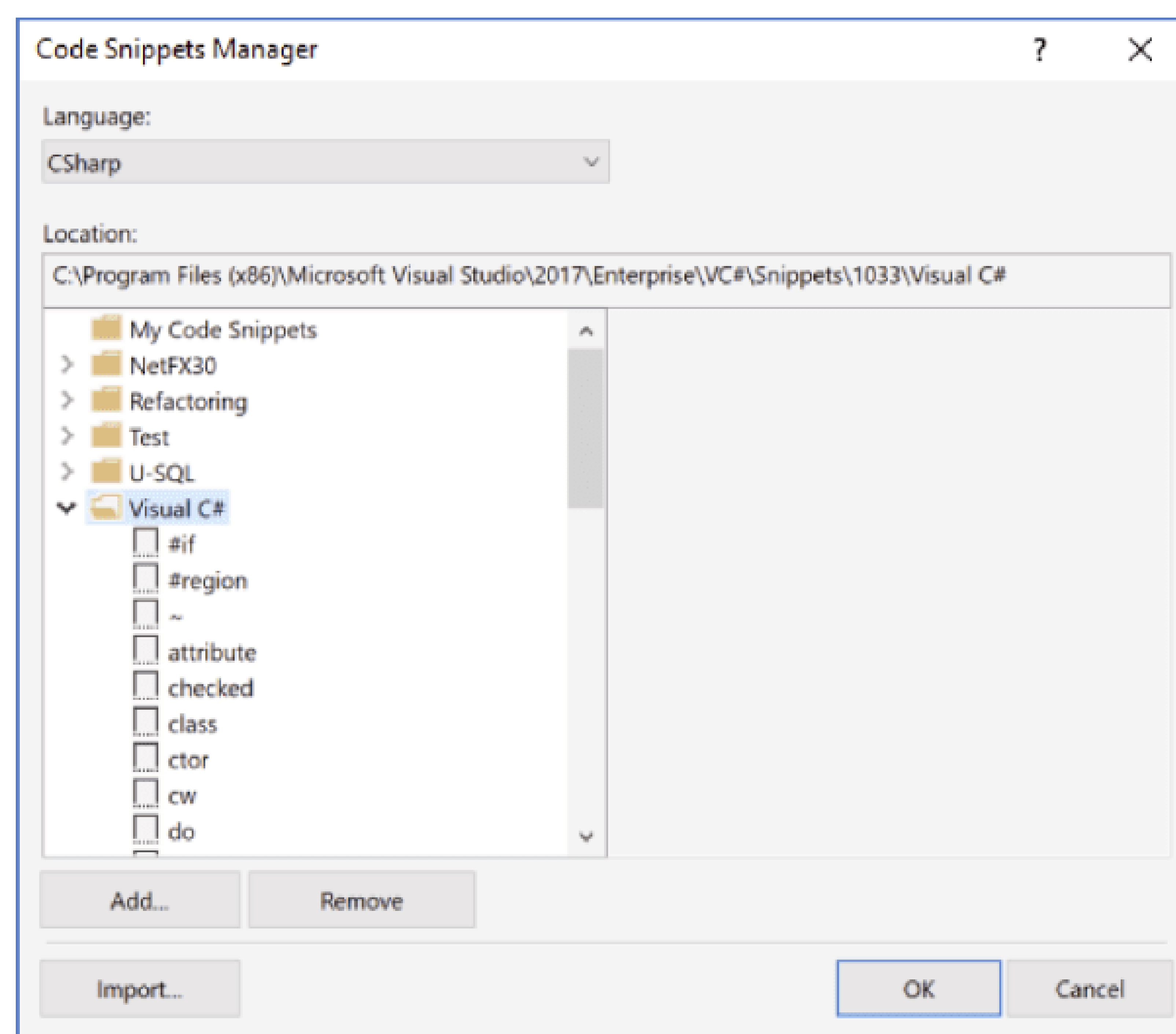



图 1-13

### 6. 构建项目

从菜单 Build | Build Solution 中编译项目。如果出现错误，Error List 窗口会显示错误和警告。但是，Output 窗口(参见图 1-14)比 Error List 窗口更可靠。有时，Error List 窗口包含了较老的缓存信息，或者当列表较大时，查找错误并不容易。Output 窗口通常为许多不同的工具提供了很好的信息。选择 View | Output，就可以打开 Output 窗口。

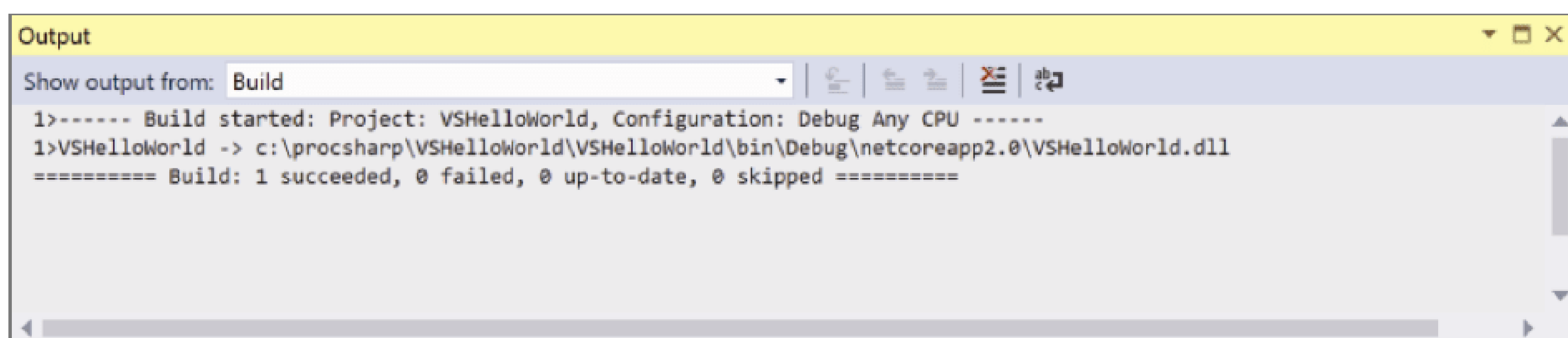



图 1-14

### 7. 运行应用程序

要运行应用程序，选择 Debug | Start Without Debugging。这将启动应用程序，并保持控制台窗口打开，直到关闭它为止。

请记住，可以选择 Debug 类别，在 Project Properties 中配置应用程序参数。

### 8. 调试

要调试应用程序，可以单击编辑器中的左侧灰色区域来创建断点(参见图 1-15)。有断点之后，就可以通过选择 Debug | Start Debugging 启动调试器。到达一个断点时，可以使用 Debug 工具栏(参见图 1-16)，以进入、结束或退出方法，也可以显示下一条语句。将鼠标悬停在变量上，可以查看当前值。还可以在 Locals 和 Watch 窗口中检查变量设置，也可以在应用程序运行时更改值。

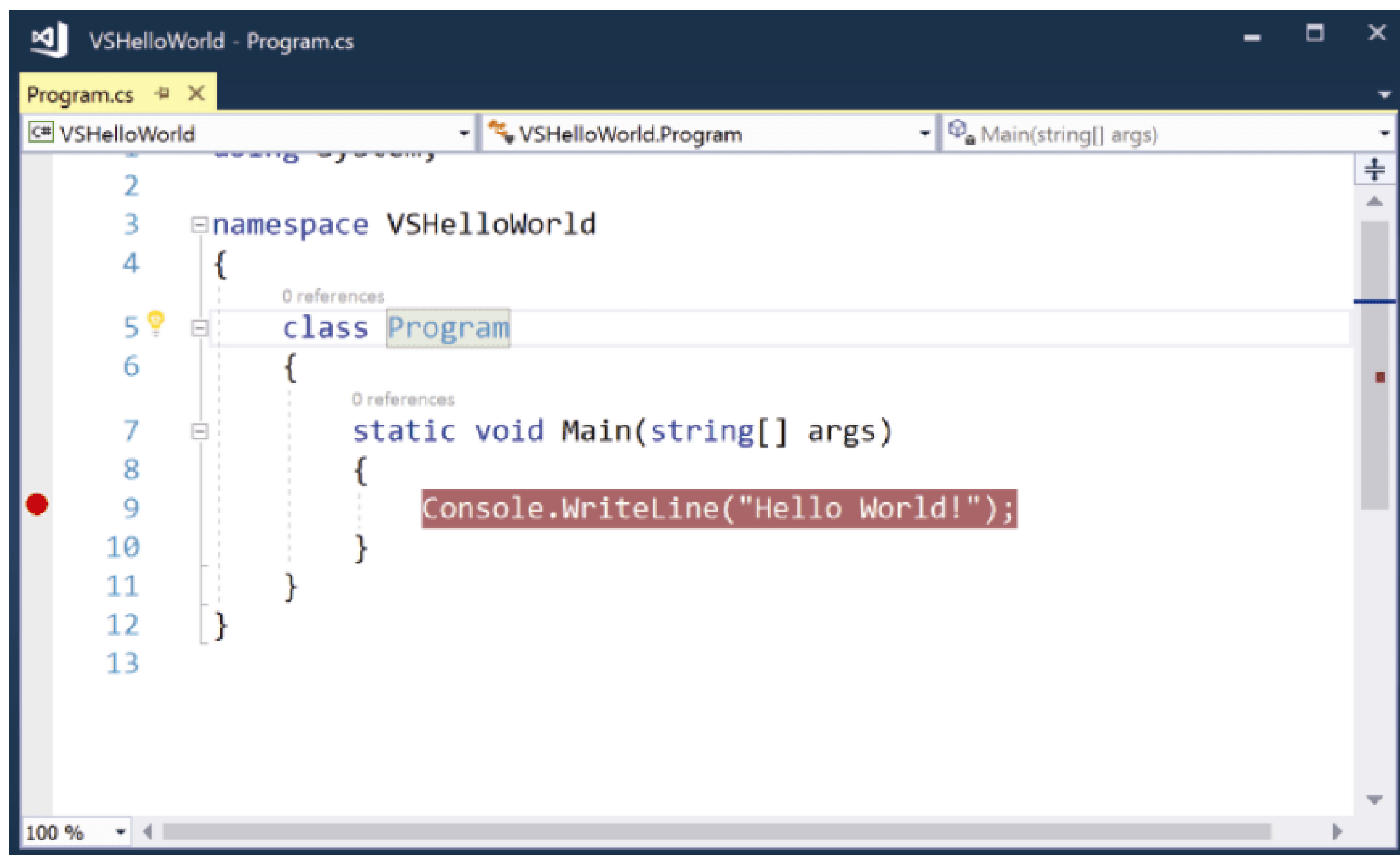



图 1-15



图 1-16

前面介绍的 Visual Studio 部分是帮助领悟本书第 1 章的最重要因素。第 18 章对 Visual Studio 2017 进行了深入的研究。

# 1.6 应用程序类型和技术

可以使用 C#创建控制台应用程序，本章的大多数示例都是控制台应用程序。对于实际的程序，控制台应用程序并不常用。使用 C#创建的应用程序可以使用与.NET 相关的许多技术。本节概述可以用 C#编写的不同类型的应用程序。

## 1.6.1 数据访问

在介绍应用程序类型之前，先看看所有应用程序类型都使用的技术：数据访问。

文件和目录可以使用简单的 API 调用来访问，但简单的 API 调用对于有些场景而言不够灵活。使用流 API 有很大的灵活性，流提供了更多的特性，例如加密或压缩。阅读器和写入器简化了流的使用。所有可用的不同选项都包含在第 22 章中。也可能以 XML 或 JSON 格式序列化完整的对象。网上附加第 2 章讨论了这些选项。

为了读取和写入数据库，可以直接使用 ADO.NET(参见第 25 章)，也可以使用抽象层：Entity Framework Core (参见第 26 章)。Entity Framework Core 提供了从对象层次结构到数据库关系的映射。

Entity Framework Core 1.0 是 Entity Framework 的完全重新设计，新名称反映了这一点。代码需要更改，把应用程序从 Entity Framework 的旧版本迁移到新版本。旧的映射变体，如 Database First 和 Model First 已被删除，因为 Code First 是更好的选择。完全重新设计也支持关系数据库和 NoSQL。Entity Framework Core 2.0 有一长串的新特性，本书将介绍这些内容。

## 1.6.2 Windows 应用程序

对于创建 Windows 应用程序，选择的技术应该是 UWP(通用 Windows 平台)。当然，当这个选项无法使用时，会有一些限制——例如，如果仍然需要支持 Windows 7 这样的旧 O/S 版本。此时，可以使用 Windows Presentation Foundation (WPF)。本书不介绍 WPF，但是可以阅读《C#高级编程(第 10 版) C# 6 & .NET Core 1.0》，它有 5 个章节专门介绍 WPF，其他章节也介绍了 WPF。

本书的一个重点是：通过 UWP 开发应用程序。与 WPF 相比，UWP 提供了更现代的 XAML 来创建用户界面。例如，数据绑定提供了一个编译后的绑定变体，在编译时会得到错误，而不是显示绑定的数据。应用程序在运行于客户端系统之前被编译为本机代码。它提供了现代的设计，现在称为微软中的流畅设计。

**注意：**

第 33 章介绍了 UWP 应用程序的创建，并介绍了 XAML、不同的 XAML 控件和应用程序的生命周期。通过支持 MVVM 模式，可以使用 WPF、UWP 和 Xamarin，利用尽可能多的公共代码来创建应用程序。这一模式在第 34 章中介绍。为了给应用程序创建酷炫的外观和风格，请务必阅读第 35 章。第 36 章深入介绍了 UWP 的一些高级特性。

## 1.6.3 Xamarin

如果 Windows 在移动电话市场上扮演更大的角色，那就太好了。然后，UWA(通用 Windows 应用程序)也会在移动电话上运行。但事实并非如此，移动电话上运行 Windows 已经是过去的事情了。然而，通过 Xamarin，你可以使用 C#和 XAML 在 iPhone 和 Android 上创建应用程序。Xamarin 提供了可以在 Android 上创建应用程序的 API，以及使用熟悉的 C#代码在 iPhone 上创建应用程序的库。

对于 Android，使用 Android Callable Wrappers (ACW)和 Managed Callable Wrappers(MCW)的映射层可以用于在.NET 代码和 Android 的 Java 运行库之间进行交互操作。在 iOS 中，Ahead of Time (AOT)编译器将托管代码编译为本地代码。

Xamarin.Forms 提供了 XAML 代码来创建用户界面，并在 Android、iOS、Windows 和 Linux 之间共享尽可能多的用户界面。XAML 只提供可以映射到所有平台的 UI 控件。为了使用平台上的特定控件，可以创建特定

于平台的渲染器。

**注意：**
用 Xamarin 和 Xamarin.Forms 开发的内容参见第 37 章。

### 1.6.4 Web 应用程序

最初引入 ASP.NET，从根本上改变了 Web 编程模型。ASP.NET Core 再次改变了它，允许使用.NET Core 提高性能和可伸缩性。这个新版本也可以在 Windows 和 Linux 系统上运行。

在 ASP.NET Core 中，不再包含 ASP.NET Web Forms(它仍然可以使用，在.NET 4.7 中更新)。

ASP.NET Core MVC 基于著名的 MVC(模型-视图-控制器)模式，更容易进行单元测试。它还允许把编写用户界面代码与 HTML、CSS、JavaScript 清晰地分离，它只在后台使用 C#。

**注意：**
第 30 章介绍了 ASP.NET Core 的基础，第 31 章继续使用 ASP.NET Core MVC 框架加固基础。

### 1.6.5 Web API

过去，SOAP 和 WCF 完成了任务，就不再需要它们了。现代应用程序利用 REST (Representational State Transfer)和 Web API。使用 ASP.NET Core 创建 Web API 是一种更容易进行通信的选项，它满足了分布式应用程序 90%以上的需求。这项技术是基于 REST 的，它为无状态、可伸缩的 Web 服务定义了指导方针和最佳实践。

客户端可以接收 JSON 或 XML 数据。JSON 和 XML 也可以格式化来使用 Open Data(OData)规范。

这个 API 的新特性更容易在 Web 客户端上使用 JavaScript、UWP 和 Xamarin。

创建 Web API 是构建微服务的好方法。构建微服务的方法定义了更小的服务，这些服务可以独立地运行和部署，可以自己控制数据存储。

为了描述服务，定义了一个新的标准：OpenAPI (https://www.openapis.org)。这个标准植根于 Swagger (https://swagger.io/)。

**注意：**
ASP.NET Core Web API、Swagger 和更多关于微服务的信息参见第 32 章。

### 1.6.6 WebHooks 和 SignalR

对于实时 Web 功能以及客户端和服务器端之间的双向通信，可以使用的 ASP.NET Core 和.NET Core 2.1 技术是 WebHooks 和 SignalR。

只要信息可用，SignalR 就允许将信息尽快推送给连接的客户。SignalR 使用 WebSocket 技术推送信息。

WebHooks 可以集成公共服务，这些服务可以调用公共 ASP.NET Core 创建的 Web API 服务。WebHooks 技术从 GitHub、Dropbox 和其他服务中接收推送通知。

**注意：**
SignalR 连接管理、连接的分组，以及 WebHooks 的授权和集成的基础知识参见网上附加第 3 章。

### 1.6.7 Microsoft Azure

现在，在考虑开发图景时不能忽视云。虽然没有专门的章节讨论云技术，但在本书的几章中都引用了 Microsoft Azure。

Microsoft Azure 提供了软件即服务(Software as a Service，SaaS)、基础设施即服务(Infrastructure as a Service，IaaS)、平台即服务(Platform as a Service，PaaS) 和函数即服务(Functions as a Service，FaaS)。有时产品介于这些类别之间。下面介绍这些 Microsoft Azure 产品。

### 1. SaaS

SaaS 提供了完整的软件，不需要处理服务器的管理和更新等。Office 365 是一个 SaaS 产品，它通过云产品使用电子邮件和其他服务。与开发人员相关的 SaaS 产品是 Visual Studio Team Server。Visual Studio Team Server 是云中的 Foundation Server，可以用作私人代码库，跟踪错误和工作项，以及构建和测试服务。第 18 章介绍了 Visual Studio 中可用的 DevOps 特征。

### 2. IaaS

另一个服务产品是 IaaS。这个服务产品提供了虚拟机。用户负责管理操作系统，维护更新。当创建虚拟机时，可以决定不同的硬件产品，从共享核心开始，到最多 128 核(编写本书时的数据，但这个数据会很快改变)。128 核、2TB 的 RAM 和 4TB 的本地 SSD 属于计算机的“M 系列”。

对于预装的操作系统，可以在 Windows、Windows Server、Linux 和预装了 SQL Server、BizTalk Server、SharePoint 和 Oracle 等的操作系统之间选择。

笔者经常给一周只需要几个小时的环境使用虚拟机，因为虚拟机按小时支付费用。如果想尝试在 Linux 上编译和运行.NET Core 程序，但没有 Linux 计算机，在 Microsoft Azure 上安装这样一个环境是很容易的。

### 3. PaaS

对于开发人员来说，Microsoft Azure 最相关的部分是 PaaS。可以为存储和读取数据而访问服务，使用应用程序服务的计算和联网功能，在应用程序中集成开发者服务。

为了在云中存储数据，可以使用关系数据存储 SQL Database。SQL Database 与 SQL Server 的本地版本大致相同。也有一些 NoSQL 解决方案，例如，Cosmos DB 有不同的存储选项，如 JSON 数据、关系或表格存储，Azure Storage 存储 blob(如图像或视频)。

应用程序服务可以用于驻留通过 ASP.NET Core 创建的 Web 应用程序和 API 应用程序。

Microsoft 还在 Microsoft Azure 中提供了 Developer Services。Developer Services 的一部分是 Visual Studio Team Services。Visual Studio Team Services 允许管理源代码，自动构建、测试和部署 CI (持续集成)。

Developer Services 的另一部分是 Application Insights。它的发布周期更短，对于获得用户如何使用应用程序的信息越来越重要。用户因为可能找不到哪些菜单，而从未使用过它们？用户在应用程序中使用什么路径来完成任务？在 Application Insights 中，可以得到良好的匿名用户信息，找出用户关于应用程序的问题，使用 DevOps 可以快速解决这些问题。

还可以使用 Cognitive Services 提供的功能处理图像，使用 Bing Search APIs，利用语言服务理解用户的看法等。

### 4. FaaS

FaaS 是云服务的一个新概念，也称为无服务器计算技术。当然，幕后总是有一个服务器。不需要为保留的 CPU 和内存付费，就像在 Web 应用程序中使用的应用程序服务一样。相反，支付的金额是基于消费的，即对活动所需要的内存的调用次数和时间付费，且有一些限制。Azure 函数是一种可以使用 FaaS 进行部署的技术。

> **注意：**
> 第 29 章介绍了跟踪特性以及如何使用 Microsoft Azure 的 Application Insights 产品。第 32 章不仅说明了如何使用 ASP. NET Core MVC 创建 Web API，也展示了如何在 Azure 函数中使用相同的服务功能。网上附加第 4 章介绍了 Microsoft Bot 服务和 Cognitive Services。

## 1.7 开发工具

第 2 章会讨论很多 C#代码，而本章的最后一部分介绍开发工具和 Visual Studio 2017 的版本。

### 1.7.1 Visual Studio Community

这个版本的 Visual Studio 是免费的，具备以前 Professional 版的功能。使用时间有许可限制。它对开源项目和培训、学术和小型专业团队是免费的。Visual Studio Express 版本以前是免费的，但该产品允许在 Visual Studio 中使用扩展。

### 1.7.2 Visual Studio Professional

这个版本比 Community 版包括更多功能，例如 CodeLens 和 Team Foundation Server，来进行源代码管理和团队协作。有了这个版本，也会得到 MSDN 订阅，其中包括微软公司的几个服务器产品，用于开发和测试。

### 1.7.3 Visual Studio Enterprise

与 Professional 版不同，这个版本包含很多测试工具，如 Web 负载和性能测试、使用 Microsoft Fakes 进行单元测试隔离，以及编码的 UI 测试(单元测试是所有 Visual Studio 版本的一部分)。通过 Code Clone 可以找到解决方案中的代码克隆。Visual Studio Enterprise 版还包含架构和建模工具，以分析和验证解决方案体系结构。

**注意：**
有了 Visual Studio 订阅，就有权免费使用 Microsoft Azure，每月具体的数量视 MSDN 订阅的类型而定。

**注意：**
第 18 章详细介绍了 Visual Studio 2017 几个特性的使用。第 28 章阐述单元测试、Web 测试和创建编码的 UI 测试。

**注意：**
本书中的一些功能，如编码的 UI 测试，需要 Visual Studio Enterprise 版。使用 Visual Studio Community 版可以完成本书的大部分内容。

### 1.7.4 Visual Studio for Mac

Visual Studio for Mac 起源于 Xamarin Studio，但现在它提供的比之前的产品多得多。例如，编辑器与 Visual Studio 共享代码，因此用户很快就会熟悉它。有了 Visual Studio for Mac，不仅可以创建 Xamarin 应用程序，还可以创建在 Windows、Linux 和 Mac 上运行的 ASP.NET Core 应用程序。在本书的许多章节中，都可以使用 Visual Studio for Mac。但介绍 UWP 的章节例外，它要求用 Windows 运行和开发应用程序。

### 1.7.5 Visual Studio Code

与其他 Visual Studio 版本相比，Visual Studio Code 是一个完全不同的开发工具。Visual Studio 2017 提供了基于项目的特性以及一组丰富的模板和工具，而 Visual Studio Code 是一个代码编辑器，几乎不支持项目管理。然而，Visual Studio Code 不仅在 Windows 上运行，也在 Linux 和 OS X 上运行。

对于本书的许多章节，可以使用 Visual Studio Code 作为开发编辑器。但不能创建 UWP 或 Xamarin 应用程序，也无法获得第 18 章介绍的特性。Visual Studio Code 代码可以用于.NET Core 控制台应用程序，以及使用.NET Core 的 ASP.NET Core 1.0 Web 应用程序。

可以从 http://code.visualstudio.com 下载 Visual Studio Code。

## 1.8 小结

本章涵盖了很多重要的技术和技术的变化。了解一些技术的历史，有助于确定新的应用程序应该使用哪些

技术，现有的应用程序应该如何处理。

.NET Framework 和.NET Core 是有差异的。本章讨论了如何在这两种环境中创建并运行“Hello World!”应用程序，但没有使用 Visual Studio。

本章阐述了公共语言运行库(CLR)的功能，介绍了用于访问数据库和创建 Windows 应用程序的技术。论述了 ASP.NET Core 的优点。

第 2 章开始讨论 C#语法，学习变量，实现程序流，把代码组织到名称空间中等内容。

# 第 2 章

# 核 心 C#

**本章要点**

- 声明变量
- 变量的初始化和作用域
- C#的预定义数据类型
- 在 C#程序中指定执行流
- 使用名称空间组织类和类型
- Main()方法
- 使用内部注释和文档编制功能
- 预处理器指令
- C#编程的推荐规则和约定

**本章源代码下载地址(wrox.com)：**

打开 www.wrox.com 的 Download Code 选项卡可下载本章源代码。源代码也可以在 CoreCSharp 目录的 https://github.com/ProfessionalCSharp/ProfessionalCSharp7 中找到。

本章代码分为以下几个主要的示例文件：

- HelloWorldApp
- VariablesSample
- VariableScopeSample
- IfStatement
- ForLoop
- NamespacesSample
- ArgumentsSample
- StringSample

## 2.1 C#基础

理解了 C#的用途后，就该学习如何使用它了。本章将介绍 C#编程的基础知识，这也是后续章节的基础。

阅读完本章后，读者就有足够的 C#知识来编写简单的程序了，但还不能使用继承或其他面向对象的特性。这些内容将在后面的几章中讨论。

### “Hello World!”程序

第 1 章解释了如何使用.NET Core CLI 工具、Visual Studio、Visual Studio for Mac 和 Visual Studio Code 编写“Hello World!”应用程序。下面解释 C#源代码。首先对 C#语法做一些一般性的解释。在 C#中，与其他 C 风格的语言一样，语句都以分号(;)结尾，并且语句可以写在多个代码行上，不需要使用续行字符。用花括号({})把语句组合为块。单行注释以两个斜杠字符开头(//)，多行注释以一个斜杠和一个星号(/*)开头，以一个星号和一个斜杠(*/)结尾。在这些方面，C#与 C++和 Java 一样，但与 Visual Basic 不同。分号和花括号使 C#代码与 Visual Basic 代码的外观差异很大。如果以前使用的是 Visual Basic，就应特别注意每条语句结尾的分号。对于新接触 C 风格语言的用户，忽略分号常常是导致编译错误的最主要原因。另一个方面是，C#区分大小写，即 myVar 与 MyVar 是两个不同的变量。

在前面的代码示例中，前几行代码与名称空间有关(如本章后面所述)，名称空间是把相关类组合在一起的方式。namespace 关键字声明了应与类相关的名称空间。其后花括号中的所有代码都被认为是在这个名称空间中。编译器在 using 语句指定的名称空间中查找没有在当前名称空间中定义但在代码中引用的类。这与 Java 中的 import 语句和 C++中的 using namespace 语句非常类似(代码文件 HelloWorldApp/Program.cs)：

```
using System;
namespace Wrox.HelloWorldApp
{
```

在 Program.cs 文件中使用“using System;声明”的原因是：下面要使用 System：System.Console 名称空间中的类 Console。using System;声明语句允许引用这个类，而忽略名称空间。可以使用下面的类调用 WriteLine 方法：

```
using System;
// ...
Console.WriteLine("Hello World!");
```

> **注意：**
> 名称空间参见本章后面的内容。

使用 using static 声明，不仅可以打开名称空间，还可以打开类的所有静态成员。声明 using static System.Console，可以调用 Console 类的 WriteLine 方法，但不使用类名：

```
using static System.Console;
// ...
WriteLine("Hello World!");
```

忽略整个 using 声明，调用 WriteLine()方法时就必须添加名称空间的名称：

```
System.Console.WriteLine("Hello World!");
```

标准的 System 名称空间包含了最常用的.NET 类型。在 C#中做的所有工作都依赖于.NET 基类，认识到这一点非常重要。在本例中，使用了 System 名称空间中的 Console 类，以写入控制台窗口。C#没有用于输入和输出的内置关键字，而是完全依赖于.NET 类。

在源代码中，声明一个类 Program。但是，因为该类位于 Wrox.HelloWorldApp 名称空间中，所以其完整的名称是 Wrox.HelloWorldApp.Program(代码文件 HelloWorldApp/Program.cs)：

```
namespace Wrox.HelloWorldApp
{
  class Program
  {
```

所有的 C#代码都必须包含在类中。类的声明包括 class 关键字，其后是类名和一对花括号。与类相关的所有代码都应放在这对花括号中。

Program 类包含一个方法 Main()。每个 C#可执行文件(如控制台应用程序、Windows 应用程序、Windows 服务和 Web 应用程序)都必须有一个入口点——Main()方法(注意，M 大写)：

```
static void Main()
{
```

在程序启动时调用该方法。该方法要么没有返回值(void)，要么返回一个整数(int)。注意，在 C#中，方法定义的格式如下：

```
[modifiers] return_type MethodName([parameters])
{
  // Method body. NB. This code block is pseudo-code.
}
```

第一个方括号中的内容表示一些可选关键字。修饰符(modifiers)用于指定用户所定义的方法的某些特性，例如可以在何处调用该方法。在本例中，Main()方法没有使用 public 访问修饰符，如果需要对 Main()方法进行单元测试，可以给它使用 public 访问修饰符。运行库不需要使用 public 访问修饰符，仍可以调用方法。运行库没有创建类的实例，调用方法时，需要修饰符 static。把返回类型设置为 void，在本例中不包含任何参数。

最后，看看代码语句：

```
Console.WriteLine("Hello World!");
```

在本例中，只调用了 System.Console 类的 WriteLine()方法，把一行文本写到控制台窗口上。WriteLine()是一个静态方法，在调用之前不需要实例化 Console 对象。

对 C#基本语法有了大致的认识后，下面就详细讨论 C#的各个方面。因为没有变量不可能编写出重要的程序，所以首先介绍 C#中的变量。

## 2.2　变量

在 C#中使用下述语法声明变量：

```
datatype identifier;
```

例如：

```
int i;
```

该语句声明 int 变量 i。实际上编译器不允许在表达式中使用这个变量，除非用一个值初始化了该变量。

声明 i 之后，就可以使用赋值运算符(=)给它赋值：

```
i = 10;
```

还可以在一行代码中(同时)声明变量，并初始化它的值：

```
int i = 10;
```

如果在一条语句中声明和初始化了多个变量，那么所有变量都具有相同的数据类型：

```
int x = 10, y =20; // x and y are both ints
```

要声明不同类型的变量，需要使用单独的语句。在一条多变量的声明中，不能指定不同的数据类型：

```
int x = 10;
bool y = true; // Creates a variable that stores true or false
int x = 10, bool y = true; // This won't compile!
```

注意上面例子中的“//”和其后的文本，它们是注释。“//”字符串告诉编译器，忽略该行后面的文本，这些文本仅为了让人更好地理解程序，它们并不是程序的一部分。本章后面会详细讨论代码中的注释。

### 2.2.1　初始化变量

变量的初始化是 C#强调安全性的另一个例子。简单地说，C#编译器需要用某个初始值对变量进行初始化，之后才能在操作中引用该变量。大多数现代编译器把没有初始化标记为警告，但 C#编译器把它当成错误来看待。

C#有两个方法可确保变量在使用前进行了初始化：

- 变量是类或结构中的字段，如果没有显式初始化，则创建这些变量时，其默认值就是 0(类和结构在后面讨论)。
- 方法的局部变量必须在代码中显式初始化，之后才能在语句中使用它们的值。此时，初始化不是在声明该变量时进行的，但编译器会通过方法检查所有可能的路径，如果检测到局部变量在初始化之前就使用了其值，就会标记为错误。

例如，在 C#中不能使用下面的语句：

```
static int Main()
{
  int d;
  Console.WriteLine(d); // Can't do this! Need to initialize d before use
  return 0;
}
```

注意在这段代码中，演示了如何定义 Main()方法，使之返回一个 int 类型而不是 void 类型的数据。

在编译这些代码时，会得到下面的错误消息：

```
Use of unassigned local variable 'd'
```

考虑下面的语句：

```
Something objSomething;
```

在 C#中，这行代码仅会为 Something 对象创建一个引用，但这个引用还没有指向任何对象。对该变量调用方法或属性会导致错误。

在 C#中实例化一个引用对象，需要使用 new 关键字。如上所述，创建一个引用，使用 new 关键字把该引用指向存储在堆上的一个对象：

```
objSomething = new Something(); // This creates a Something object on the heap
```

### 2.2.2 类型推断

类型推断使用 var 关键字。声明变量的语法有些变化：使用 var 关键字替代实际的类型。编译器可以根据变量的初始化值“推断”变量的类型。例如：

```
var someNumber = 0;
```

就变成：

```
int someNumber = 0;
```

即使 someNumber 从来没有声明为 int，编译器也可以确定，只要 someNumber 在其作用域内，就是 int 类型。编译后，上面两个语句是等价的。

下面是另一个小例子(代码文件 VariablesSample/Program.cs )：

```
using System;
namespace Wrox
{
  class Program
  {
    static void Main()
    {
      var name = "Bugs Bunny";
      var age = 25;
      var isRabbit = true;
      Type nameType = name.GetType();
      Type ageType = age.GetType();
      Type isRabbitType = isRabbit.GetType();
      Console.WriteLine($"name is of type {nameType}");
      Console.WriteLine($"age is of type {ageType}");
      Console.WriteLine($"isRabbit is of type {isRabbitType}");
    }
  }
}
```

这个程序的输出如下：

```
name is of type System.String
age is of type System.Int32
isRabbit is of type System.Boolean
```

需要遵循以下一些规则：

- 变量必须初始化。否则，编译器就没有推断变量类型的依据。
- 初始化器不能为空。
- 初始化器必须放在表达式中。
- 不能把初始化器设置为一个对象，除非在初始化器中创建了一个新对象。

第 3 章在讨论匿名类型时将详细探讨这些规则。

声明了变量且推断出类型后，就不能再改变变量的类型了。变量的类型确定后，对该变量进行任何赋值时，其强类型化规则必须以推断出的类型为基础。

## 2.2.3 变量的作用域

变量的作用域是可以访问该变量的代码区域。一般情况下，确定作用域遵循以下规则：

- 只要类的局部变量在某个作用域内，其字段(也称为成员变量)也在该作用域内。
- 局部变量存在于表示声明该变量的块语句或方法结束的右花括号之前的作用域内。
- 在 for、while 或类似语句中声明的局部变量存在于该循环体内。

### 1. 局部变量的作用域冲突

大型程序常常在不同部分为不同的变量使用相同的变量名。只要变量的作用域是程序的不同部分，就不会有问题，也不会产生多义性。但要注意，同名的局部变量不能在同一作用域内声明两次。例如，不能使用下面的代码：

```
int x = 20;
// some more code
int x = 30;
```

考虑下面的代码示例(代码文件 VariableScopeSample/Program.cs )：

```
using System;
namespace VariableScopeSample
{
  class Program
  {
    static int Main()
    {
      for (int i = 0; i < 10; i++)
      {
        Console.WriteLine(i);
      } // i goes out of scope here

      // We can declare a variable named i again, because
      // there's no other variable with that name in scope
      for (int i = 9; i >= 0; i -)
      {
        Console.WriteLine(i);
      } // i goes out of scope here.

      return 0;
    }
  }
}
```

这段代码很简单，使用两个 for 循环打印 0~9 的数字，再逆序打印 0~9 的数字。重要的是在同一个方法中，代码中的变量 i 声明了两次。可以这么做的原因是 i 在两个相互独立的循环内部声明，所以每个变量 i 对于各自的循环来说是局部变量。

下面是另一个例子(代码文件 VariableScopeSample2/Program.cs )：

```
static int Main()
```

```
{
  int j = 20;
  for (int i = 0; i < 10; i++)
  {
    int j = 30; // Can't do this - j is still in scope
    Console.WriteLine(j + i);
  }
  return 0;
}
```

如果试图编译它，就会产生如下错误：

```
error CS0136: A local variable named 'j' cannot be declared in
this scope because that name is used in an enclosing local scope
to define a local or parameter
```

其原因是：变量 j 是在for循环开始之前定义的，在执行for循环时仍处于其作用域内，直到Main()方法结束执行后，变量 j 才超出作用域。第2个 j (不合法)虽然在循环的作用域内，但作用域嵌套在Main()方法的作用域内。因为编译器无法区分这两个变量，所以不允许声明第二个变量。

#### 2. 字段和局部变量的作用域冲突

某些情况下，可以区分名称相同(尽管其完全限定名不同)、作用域相同的两个标识符。此时编译器允许声明第二个变量。原因是 C#在变量之间有一个基本的区分，它把在类型级别声明的变量看成字段，而把在方法中声明的变量看成局部变量。

考虑下面的代码片段(代码文件 VariableScopeSample3/Program.cs)：

```
using System;
namespace Wrox
{
  class Program
  {
    static int j = 20;
    static void Main()
    {
      int j = 30;
      Console.WriteLine(j);
      return;
    }
  }
}
```

虽然在 Main()方法的作用域内声明了两个变量 j，这段代码也会编译：一个是在类级别上定义的 j，在类 Program 删除前(在本例中，是 Main()方法终止，程序结束时)是不会超出作用域的；一个是在 Main()中定义的 j。这里，在 Main()方法中声明的新变量 j 隐藏了同名的类级别变量，所以在运行这段代码时，会显示数字 30。

但是，如果要引用类级别变量，该怎么办？可以使用语法 object.fieldname，在对象的外部引用类或结构的字段。在上面的例子中，访问静态方法中的一个静态字段，所以不能使用类的实例，只能使用类本身的名称：

```
// ...
static void Main()
{
  int j = 30;
  Console.WriteLine(j);
  Console.WriteLine(Program.j);
}
// ...
```

如果要访问实例字段(该字段属于类的一个特定实例)，就需要使用 this 关键字。

### 2.2.4 常量

顾名思义，常量是其值在使用过程(生命周期)中不会发生变化的变量。在声明和初始化变量时，在变量的前面加上关键字 const，就可以把该变量指定为一个常量：

```
const int a = 100; // This value cannot be changed.
```

常量具有如下特点：

- 常量必须在声明时初始化。指定了其值后，就不能再改写了。
- 常量的值必须能在编译时用于计算。因此，不能用从变量中提取的值来初始化常量。如果需要这么做，应使用只读字段(详见第 3 章)。
- 常量总是隐式静态的。但注意，不必(实际上，是不允许)在常量声明中包含修饰符 static。

在程序中使用常量至少有 3 个好处：

- 由于使用易于读取的名称(名称的值易于理解)替代了较难读取的数字和字符串，常量使程序变得更易于阅读。
- 常量使程序更易于修改。例如，在 C#程序中有一个 SalesTax 常量，该常量的值为 6%。如果以后销售税率发生变化，把新值赋给这个常量，就可以修改所有的税款计算结果，而不必查找整个程序去修改税率为 0.06 的每个项。
- 常量更容易避免程序出现错误。如果在声明常量的位置以外的某个地方将另一个值赋给常量，编译器就会标记错误。

## 2.3 预定义数据类型

前面介绍了如何声明变量和常量，下面要详细讨论 C#中可用的数据类型。与其他语言相比，C#对其可用的类型及其定义有更严格的描述。

### 2.3.1 值类型和引用类型

在开始介绍 C#中的数据类型之前，理解 C#把数据类型分为两种非常重要：

- 值类型
- 引用类型

下面几节将详细介绍值类型和引用类型的语法。从概念上看，其区别是值类型直接存储其值，而引用类型存储对值的引用。

这两种类型存储在内存的不同地方：值类型存储在堆栈(stack)中，而引用类型存储在托管堆(managed heap)上。注意区分某个类型是值类型还是引用类型，因为这会有不同的影响。

例如，int 是值类型，这表示下面的语句会在内存的两个地方存储值 20：

```
// i and j are both of type int
i = 20;
j = i;
```

但考虑下面的代码。这段代码假定已经定义了类 Vector，Vector 是一个引用类型，它有一个 int 类型的成员变量 Value：

```
Vector x, y;
x = new Vector();
x.Value = 30; // Value is a field defined in Vector class
y = x;
Console.WriteLine(y.Value);
y.Value = 50;
Console.WriteLine(x.Value);
```

要理解的重要一点是：在执行这段代码后，只有一个 Vector 对象。x 和 y 都指向包含该对象的内存位置。因为 x 和 y 是引用类型的变量，声明这两个变量只保留了一个引用——而不会实例化给定类型的对象。两种情况下都不会真正创建对象。要创建对象，就必须使用 new 关键字，如上所示。因为 x 和 y 引用同一个对象，所以对 x 的修改会影响 y，反之亦然。因此上面的代码会显示 30 和 50。

如果变量是一个引用，就可以把其值设置为 null，表示它不引用任何对象：

```
y = null;
```

如果将引用设置为 null，显然就不可能对它调用任何非静态的成员函数或字段，这么做会在运行期间抛出一个异常。

**注意：**
C# 8 计划支持不可空的引用类型。这些类型的变量需要使用非空值进行初始化。允许使用空值的引用类型显式地要求声明为可空的引用类型。

在 C#中，基本数据类型(如 bool 和 long)都是值类型。如果声明一个 bool 变量，并给它赋予另一个 bool 变量的值，在内存中就会有两个 bool 值。如果以后修改第一个 bool 变量的值，第二个 bool 变量的值也不会改变。这些类型是通过值来复制的。

相反，大多数更复杂的 C#数据类型，包括我们自己声明的类，都是引用类型。它们分配在堆中，其生存期可以跨多个函数调用，可以通过一个或几个别名来访问。CLR 实现一种精细的算法，来跟踪哪些引用变量仍是可以访问的，哪些引用变量已经不能访问了。CLR 会定期删除不能访问的对象，把它们占用的内存返回给操作系统。这是通过垃圾收集器实现的。

把基本类型(如 int 和 bool)规定为值类型，而把包含许多字段的较大类型(通常在有类的情况下)规定为引用类型，C#设计这种方式是为了得到最佳性能。如果要把自己的类型定义为值类型，就应把它声明为一个结构。

**注意：**
原始数据类型的布局通常与本机布局保持一致。因此可能在托管代码和本机代码之间共享相同的内存。

## 2.3.2 .NET 类型

数据类型的 C#关键字(如 int、short 和 string)从编译器映射到.NET 数据类型。例如，在 C#中声明一个 int 类型的数据时，声明的实际上是.NET struct:System.Int32 的一个实例。这听起来似乎很深奥，但其意义深远：这表示在语法上，可以把所有的基本数据类型看成支持某些方法的类。例如，要把 int i 转换为 string 类型，可以编写下面的代码：

```
string s = i.ToString();
```

应该强调的是，在这种便利语法的背后，类型实际上仍存储为基本类型。基本类型在概念上用 C#结构表示，所以肯定没有性能损失。

下面看看 C#中定义的内置类型。我们将列出每个类型，以及它们的定义和对应.NET 类型的名称。C#有 15 个预定义类型，其中 13 个是值类型，两个是引用类型(string 和 object)。

## 2.3.3 预定义的值类型

内置的.NET 值类型表示基本类型，如整型和浮点类型、字符类型以及布尔类型。

### 1. 整型

C#支持 8 个预定义的整型类型，如表 2-1 所示。

表 2-1

| 名 称 | .NET 类 型 | 说 明 | 范围(最小～最大) |
|---|---|---|---|
| sbyte | System.SByte | 8 位有符号的整数 | –128～127 ($-2^7$～$2^7-1$) |
| short | System.Int16 | 16 位有符号的整数 | –32 768～32 767 ($-2^{15}$～$2^{15}-1$) |
| int | System.Int32 | 32 位有符号的整数 | –2 147 483 648～2 147 483 647 ($-2^{31}$～$2^{31}-1$) |
| long | System.Int64 | 64 位有符号的整数 | –9 223 372 036 854 775 808～9 223 372 036 854 775 807 ($-2^{63}$～$2^{63}-1$) |

(续表)

| 名　　称 | .NET 类 型 | 说　　明 | 范围(最小～最大) |
|---|---|---|---|
| byte | System.Byte | 8 位无符号的整数 | 0～255 (0～$2^8$–1) |
| ushort | System.UInt16 | 16 位无符号的整数 | 0～65 535 (0～$2^{16}$–1) |
| uint | System.UInt32 | 32 位无符号的整数 | 0～4 294 967 295 (0～$2^{32}$–1) |
| ulong | System.UInt64 | 64 位无符号的整数 | 0～18 446 744 073 709 551 615 (0～$2^{64}$–1) |

有些 C#类型的名称与 C++和 Java 类型一致，但定义不同。例如，在 C#中，int 总是 32 位有符号的整数。而在 C++中，int 是有符号的整数，但其位数取决于平台(在 Windows 上是 32 位)。在 C#中，所有的数据类型都以与平台无关的方式定义，以备将来从 C#和.NET 迁移到其他平台上。

byte 是 0~255(包括 255)的标准 8 位类型。注意，在强调类型的安全性时，C#认为 byte 类型和 char 类型完全不同，它们之间的编程转换必须显式请求。还要注意，与整数中的其他类型不同，byte 类型在默认状态下是无符号的，其有符号的版本有一个特殊的名称 sbyte。

在.NET 中，short 不再很短，现在它有 16 位长。int 类型更长，有 32 位。long 类型最长，其值有 64 位。所有整数类型的变量都能被赋予十进制或十六进制的值，后者需要 0x 前缀：

```
long x = 0x12ab;
```

如果对一个 int、uint、long 或 ulong 类型的整数没有任何显式的声明，则该变量默认为 int 类型。

为了把输入的值指定为其他整数类型，可以在数字后面加上如下字符：

```
uint ui = 1234U;
long l = 1234L;
ulong ul = 1234UL;
```

也可以使用小写字母 u 和 l，但后者会与整数 1 混淆。

**数字分隔符**

C# 7 提供了数字分隔符。这些分隔符有助于提高可读性，且不添加任何功能。例如，可以向数字添加下划线，如下面的代码片段所示(代码文件 UsingNumbers/Program.cs)：

```
long l1 = 0x123_4567_89ab_cedf;
```

用作分隔符的下划线被编译器忽略。对于前面的示例，每次从右边读取 16 位(或 4 个十六进制字符)，就添加一个数字分隔符。结果比另一种更容易读懂：

```
long l2 = 0x123456789abcedf;
```

因为编译器只会忽略下划线，所以用户要负责确保可读性。可以在任何位置放置下划线，并需要确保它有助于提高可读性，而不是如本例所示：

```
long l3 = 0x12345_6789_abc_ed_f;
```

允许把数字分隔符放在任何位置都是有用的，因为这允许把它用于不同的情况——例如，使用十六进制或八进制值，或者分离协议所需的不同位(如下一节所示)。

**注意：**
数字分隔符是 C# 7 新增的内容。C# 7.0 不允许前置数字分隔符，在值之前(和前缀之后)有分隔符。前导数字分隔符可以在 C# 7.2 中使用。

**二进制值**

除了提供数字分隔符，C# 7 还便于把二进制值分配给整数类型。如果在变量值前面加上 0b 字面值作为前缀，只允许使用 0 和 1。只有二进制值可以分配给变量，如下面的代码片段所示(代码文件 UsingNumbers/Program.cs)：

```
uint binary1 = 0b1111_1110_1101_1100_1011_1010_1001_1000;
```

前面的代码片段使用一个 32 位的无符号 int。数字分隔符对二进制值的可读性有很大帮助。这段代码把二进制值分隔为 4 位一组。记住，也可以用十六进制记数法：

```
uint hex1 = 0xfedcba98;
```

使用八进制记数法时，每三位使用一个分隔符会有所帮助，在 0(二进制为 000)和 7(二进制为 111)之间使用字符：

```
uint binary2 = 0b111_110_101_100_011_010_001_000;
```

下面的示例展示了如何定义可以在二进制协议中使用的值，其中两个位定义了最右边的部分，6 位在下一部分中，最后两个部分有 4 位来完成 16 位：

```
ushort binary3 = 0b1111_0000_101010_11;
```

记住，为需要的位数使用正确的整数类型：ushort 用于 16 位，uint 用于 32 位，ulong 用于 64 位。

**注意：**
第 6 和第 11 章介绍了二进制数据的更多信息。

**注意：**
二进制字面值是 C# 7 的新增功能。

### 2. 浮点类型

C#提供了许多整型数据类型，也支持浮点类型，如表 2-2 所示。

表 2-2

| 名　称 | .NET 类 型 | 说　明 | 位　数 | 范围(大致) |
|---|---|---|---|---|
| float | System.Single | 32 位单精度浮点数 | 7 | $\pm1.5\times10^{245}$～$\pm3.4\times10^{38}$ |
| double | System.Double | 64 位双精度浮点数 | 15/16 | $\pm5.0\times10^{2324}$～$\pm1.7\times10^{308}$ |

float 数据类型用于较小的浮点值，因为它要求的精度较低。double 数据类型比 float 数据类型大，提供的精度也大一倍(15 位)。

如果在代码中对某个非整数值(如 12.3)硬编码，则编译器一般假定该变量是 double。如果想指定该值为 float，可以在其后加上字符 F(或 f)：

```
float f = 12.3F;
```

### 3. decimal 类型

decimal 类型表示精度更高的浮点数，如表 2-3 所示。

表 2-3

| 名　称 | .NET 类 型 | 说　明 | 位　数 | 范围(大致) |
|---|---|---|---|---|
| decimal | System.Decimal | 128 位高精度十进制数表示法 | 28 | $\pm1.0\times10^{228}$～$\pm7.9\times10^{28}$ |

.NET 和 C#数据类型的一个重要优点是提供了一种专用类型进行财务计算，这就是 decimal 类型。使用 decimal 类型提供的 28 位的方式取决于用户。换言之，可以用较大的精确度(带有美分)来表示较小的美元值，也可以在小数部分用更多的舍入来表示较大的美元值。但应注意，decimal 类型不是基本类型，所以在计算时使用该类型会有性能损失。

要把数字指定为 decimal 类型而不是 double、float 或整数类型，可以在数字的后面加上字符 M(或 m)，如下所示：

```
decimal d = 12.30M;
```

#### 4. bool 类型

C#的 bool 类型用于包含布尔值 true 或 false，如表 2-4 所示。

表 2-4

| 名　称 | .NET 类 型 | 说　明 | 位　数 | 值 |
|---|---|---|---|---|
| bool | System.Boolean | 表示 true 或 false | NA | true 或 false |

bool 值和整数值不能相互隐式转换。如果变量(或函数的返回类型)声明为 bool 类型，就只能使用值 true 或 false。如果试图使用 0 表示 false，非 0 值表示 true，就会出错。

#### 5. 字符类型

为了保存单个字符的值，C#支持 char 数据类型，如表 2-5 所示。

表 2-5

| 名　称 | .NET 类 型 | 值 |
|---|---|---|
| char | System.Char | 表示一个 16 位的(Unicode)字符 |

char 类型的字面量是用单引号括起来的，如'A'。如果把字符放在双引号中，编译器会把它看成字符串，从而产生错误。

除了把 char 表示为字符字面量之外，还可以用 4 位十六进制的 Unicode 值(如'\u0041')、带有强制类型转换的整数值(如(char)65)或十六进制数(如'\x0041')表示它们。它们还可以用转义序列表示，如表 2-6 所示。

表 2-6

| 转 义 序 列 | 字　符 |
|---|---|
| \' | 单引号 |
| \" | 双引号 |
| \\ | 反斜杠 |
| \0 | 空 |
| \a | 警告 |
| \b | 退格 |
| \f | 换页 |
| \n | 换行 |
| \r | 回车 |
| \t | 水平制表符 |
| \v | 垂直制表符 |

#### 6. 数字的字面值

表 2-7 总结了可以用于数字的字面值。该表重复前几节的字面值，将它们集中在一个地方。

表 2-7

| 字 面 值 | 位　置 | 说　明 |
|---|---|---|
| U | 后缀 | unsigned int |
| L | 后缀 | long |

(续表)

| 字 面 值 | 位 置 | 说 明 |
| --- | --- | --- |
| UL | 后缀 | unsigned long |
| F | 后缀 | float |
| M | 后缀 | 十进制 (货币) |
| 0x | 前缀 | 十六进制数字，允许使用 0~F |
| 0b | 前缀 | 二进制数字，只允许使用 0 和 1 |
| true | NA | 布尔值 |
| False | NA | 布尔值 |

### 2.3.4 预定义的引用类型

C#支持两种预定义的引用类型：object 和 string，如表 2-8 所示。

表 2-8

| 名 称 | .NET 类 型 | 说 明 |
| --- | --- | --- |
| object | System.Object | 根类型，其他类型都是从它派生而来的(包括值类型) |
| string | System.String | Unicode 字符串 |

#### 1. object 类型

许多编程语言和类层次结构都提供了根类型，层次结构中的其他对象都从它派生而来。C#和.NET 也不例外。在 C#中，object 类型就是最终的父类型，所有内置类型和用户定义的类型都从它派生而来。这样，object 类型就可以用于两个目的：

- 可以使用 object 引用来绑定任何特定子类型的对象。例如，第 6 章将说明如何使用 object 类型把堆栈中的值对象装箱，再移动到堆中。object 引用也可以用于反射，此时必须有代码来处理类型未知的对象。
- object 类型实现了许多一般用途的基本方法，包括 Equals()、GetHashCode()、GetType()和 ToString()。用户定义的类需要使用一种面向对象技术——重写，来提供其中一些方法的替代实现代码。例如，重写 ToString()时，要给类提供一个方法，给出类本身的字符串表示。如果类中没有提供这些方法的实现代码，编译器就会使用 object 类型中的实现代码，它们在类上下文中的执行不一定正确。

后面将详细讨论 object 类型。

#### 2. string 类型

C#有 string 关键字，在遮罩下转换为.NET 类 System.String。有了它，像字符串连接和字符串复制这样的操作就很简单了：

```
string str1 = "Hello ";
string str2 = "World";
string str3 = str1 + str2; // string concatenation
```

尽管这是一个值类型的赋值，但 string 是一个引用类型。string 对象被分配在堆上，而不是栈上。因此，当把一个字符串变量赋予另一个字符串时，会得到对内存中同一个字符串的两个引用。但是，string 与引用类型的常见行为有一些区别。例如，字符串是不可改变的。修改其中一个字符串，就会创建一个全新的 string 对象，而另一个字符串不发生任何变化。考虑下面的代码(代码文件 StringSample/Program.cs)：

```
using System;

class Program
{
  static void Main()
  {
    string s1 = "a string";
```

```
    string s2 = s1;
    Console.WriteLine("s1 is " + s1);
    Console.WriteLine("s2 is " + s2);
    s1 = "another string";
    Console.WriteLine("s1 is now " + s1);
    Console.WriteLine("s2 is now " + s2);
  }
}
```

其输出结果为：

```
s1 is a string
s2 is a string
s1 is now another string
s2 is now a string
```

改变 s1 的值对 s2 没有影响，这与我们期待的引用类型正好相反。当用值 a string 初始化 s1 时，就在堆上分配了一个新的 string 对象。在初始化 s2 时，引用也指向这个对象，所以 s2 的值也是 a string。但是当现在要改变 s1 的值时，并不会替换原来的值，而是在堆上为新值分配一个新对象。s2 变量仍指向原来的对象，所以它的值没有改变。这实际上是运算符重载的结果，运算符重载详见第 6 章。基本上，string 类已实现，其语义遵循一般的、直观的字符串规则。

字符串字面量放在双引号中("...")；如果试图把字符串放在单引号中，编译器就会把它当成 char 类型，从而抛出错误。C#字符串和 char 一样，可以包含 Unicode 和十六进制数转义序列。因为这些转义序列以一个反斜杠开头，所以不能在字符串中使用没有经过转义的反斜杠字符，而需要用两个反斜杠字符(\\)表示它：

```
string filepath = "C:\\ProCSharp\\First.cs";
```

**警告：**

注意，对目录使用反斜杠(\)并使用 C:将应用程序限制为 Windows 操作系统。Windows 和 Linux 都可以使用斜杠(/)分隔目录。第 22 章提供了关于如何处理 Windows 和 Linux 上的文件和目录的详细信息。

输入两个反斜杠字符会令人迷惑。幸好，C#提供了替代方式。可以在字符串字面量的前面加上字符@，在这个字符后的所有字符都看成其原来的含义——它们不会解释为转义字符：

```
string filepath = @"C:\ProCSharp\First.cs";
```

甚至允许在字符串字面量中包含换行符：

```
string jabberwocky = @"'Twas brillig and the slithy toves
Did gyre and gimble in the wabe.";
```

那么 jabberwocky 的值就是：

```
'Twas brillig and the slithy toves
Did gyre and gimble in the wabe.
```

C#定义了一种新的字符串插值格式，用$前缀来标记。可以使用字符串插值格式，改变前面演示字符串连接的代码片段。对字符串加上$前缀，就允许把花括号放在包含一个变量或代码表达式的字符串中。变量或代码表达式的结果放在字符串中花括号所在的位置：

```
public static void Main()
{
  string s1 = "a string";
  string s2 = s1;
  Console.WriteLine($"s1 is {s1}");
  Console.WriteLine($"s2 is {s2}");
  s1 = "another string";
  Console.WriteLine($"s1 is now {s1}");
  Console.WriteLine($"s2 is now {s2}");
}
```

**注意：**

字符串和字符串插值功能参见第 9 章。

# 2.4 程序流控制

本节将介绍 C#语言最基本的重要语句：控制程序流的语句。它们不是按代码在程序中的排列位置顺序执行的。

## 2.4.1 条件语句

条件语句可以根据条件是否满足或根据表达式的值来控制代码的执行分支。C#有两个控制代码的分支的结构：if 语句，测试特定条件是否满足；switch 语句，比较表达式和多个不同的值。

### 1. if 语句

对于条件分支，C#继承了 C 和 C++的 if...else 结构。对于用过程语言编程的人，其语法非常直观：

```
if (condition)
  statement(s)
else
  statement(s)
```

如果在条件中要执行多个语句，就需要用花括号({ ... })把这些语句组合为一个块(这也适用于其他可以把语句组合为一个块的 C#结构，如 for 和 while 循环)。

```
bool isZero;
if (i == 0)
{
  isZero = true;
  Console.WriteLine("i is Zero");
}
else
{
  isZero = false;
  Console.WriteLine("i is Non-zero");
}
```

还可以单独使用 if 语句，不加最后的 else 语句。也可以合并 else if 子句，测试多个条件(代码文件 IfStatement/Program.cs)。

```
using System;
namespace Wrox
{
  class Program
  {
    static void Main()
    {
      Console.WriteLine("Type in a string");
      string input;
      input = Console.ReadLine();
      if (input == "")
      {
        Console.WriteLine("You typed in an empty string.");
      }
      else if (input.Length < 5)
      {
        Console.WriteLine("The string had less than 5 characters.");
      }
      else if (input.Length < 10)
      {
        Console.WriteLine(
          "The string had at least 5 but less than 10 Characters.");
      }
      Console.WriteLine("The string was " + input);
    }
  }
}
```

添加到 if 子句中的 else if 语句的个数不受限制。

注意，在上面的例子中，声明了一个字符串变量 input，让用户在命令行中输入文本，把文本填充到 input 中，然后测试该字符串变量的长度。代码还显示了在 C#中如何进行字符串处理。例如，要确定 input 的长度，可以使用 input.Length。

对于 if 语句，要注意的一点是如果条件分支中只有一条语句，就不需要使用花括号：

```
if (i == 0)
  Console.WriteLine("i is Zero"); // This will only execute if i == 0
  Console.WriteLine("i can be anything"); // Will execute whatever the
// value of i
```

但是，为了保持一致，许多程序员只要使用 if 语句，就加上花括号。

**提示：**

在 if 语句中不使用花括号，可能在维护代码时导致错误。无论 if 语句返回 true 还是 false，都常常给 if 语句添加第二条语句。每次都使用花括号，就可以避免这个编码错误。

使用 if 语句的一个指导原则是只有语句和 if 语句写在同一行上，才不允许程序员使用花括号。遵守这条指导原则，程序员就不太可能在添加第二条语句时不添加花括号。

前面介绍的 if 语句还演示了用于比较数值的一些 C#运算符。特别注意，C#使用“==”对变量进行等于比较。此时不要使用“=”，一个“=”用于赋值。

在 C#中，if 子句中的表达式必须等于布尔值(Boolean)。不能直接测试整数(如从函数中返回的值)，而必须显式地把返回的整数转换为布尔值 true 或 false，例如，将值与 0 或 null 进行比较：

```
if (DoSomething() != 0)
{
  // Non-zero value returned
}
else
{
  // Returned zero
}
```

### 2. switch 语句

switch…case 语句适合于从一组互斥的可执行分支中选择一个执行分支。其形式是 switch 参数的后面跟一组 case 子句。如果 switch 参数中表达式的等于某个 case 子句旁边的某个值，就执行该 case 子句中的代码。此时不需要使用花括号把语句组合到块中，只需要使用 break 语句标记每段 case 代码的结尾即可。也可以在 switch 语句中包含一条 default 子句，如果表达式不等于任何 case 子句的值，就执行 default 子句的代码。下面的 switch 语句测试 integerA 变量的值：

```
switch (integerA)
{
  case 1:
    Console.WriteLine("integerA = 1");
    break;
  case 2:
    Console.WriteLine("integerA = 2");
    break;
  case 3:
    Console.WriteLine("integerA = 3");
    break;
  default:
    Console.WriteLine("integerA is not 1, 2, or 3");
    break;
}
```

注意 case 值必须是常量表达式，不允许使用变量。

C 和 C++程序员应该很熟悉 switch…case 语句，而 C#的 switch…case 语句更安全。特别是它禁止几乎所有 case 中的失败条件。如果激活了块中靠前的一条 case 子句，后面的 case 子句就不会被激活，除非使用 goto 语句特别标记也要激活后面的 case 子句。编译器会把没有 break 语句的 case 子句标记为错误，从而强制实现这一约束：

```
Control cannot fall through from one case label ('case 2:') to another
```

在有限的几种情况下，这种失败是允许的，但大多数情况下，我们不希望出现这种失败，而且这会导致出现很难察觉的逻辑错误。让代码正常工作，而不是出现异常，这样不是更好吗？

但在使用 goto 语句时，会在 switch…cases 中重复出现失败。如果确实想这么做，就应重新考虑设计方案了。下面的代码说明了如何使用 goto 模拟失败，得到的代码会非常混乱：

```
// assume country and language are of type string
switch(country)
{
  case "America":
    CallAmericanOnlyMethod();
    goto case "Britain";
  case "France":
    language = "French";
    break;
  case "Britain":
    language = "English";
    break;
}
```

但有一种例外情况。如果一条 case 子句为空，就可以从这条 case 子句跳到下一条 case 子句，这样就可以用相同的方式处理两条或多条 case 子句(不需要 goto 语句)。

```
switch(country)
{
  case "au":
  case "uk":
  case "us":
    language = "English";
    break;
  case "at":
  case "de":
    language = "German";
    break;
}
```

在 C#中，switch 语句的一个有趣的地方是 case 子句的顺序是无关紧要的，甚至可以把 default 子句放在最前面！因此，任何两条 case 都不能相同。这包括值相同的不同常量，所以不能这样编写：

```
// assume country is of type string
const string england = "uk";
const string britain = "uk";
switch(country)
{
  case england:
  case britain: // This will cause a compilation error.
    language = "English";
    break;
}
```

上面的代码还说明了 C#中的 switch 语句与 C++中的 switch 语句的另一个不同之处：在 C#中，可以把字符串用作测试的变量。

**注意：**

*在 C# 7 中， switch 语句通过模式匹配得到了增强。使用模式匹配，case 的排序变得很重要。第 13 章包含使用模式匹配的 switch 语句的更多信息。*

## 2.4.2 循环

C#提供了 4 种不同的循环机制(for、while、do...while 和 foreach)，在满足某个条件之前，可以重复执行代码块。

### 1. for 循环

C#的 for 循环提供的迭代循环机制是在执行下一次迭代前，测试是否满足某个条件，其语法如下：

其中：

- initializer 是指在执行第一次循环前要计算的表达式(通常把一个局部变量初始化为循环计数器)。
- condition 是在每次循环的新迭代之前要测试的表达式(它必须等于 true，才能执行下一次迭代)。

- iterator 是每次迭代完要计算的表达式(通常是递增循环计数器)。

当 condition 等于 false 时，迭代停止。

for 循环是所谓的预测试循环，因为循环条件是在执行循环语句前计算的，如果循环条件为假，循环语句就根本不会执行。

for 循环非常适合用于一条语句或语句块重复执行预定的次数。下面的例子就是 for 循环的典型用法，这段代码输出 0~99 的整数：

```
for (int i = 0; i < 100; i = i + 1)
{
  Console.WriteLine(i);
}
```

这里声明了一个 int 类型的变量 i，并把它初始化为 0，用作循环计数器。接着测试它是否小于 100。因为这个条件等于 true，所以执行循环中的代码，显示值 0。然后给该计数器加 1，再次执行该过程。当 i 等于 100 时，循环停止。

实际上，上述编写循环的方式并不常用。C#在给变量加 1 时有一种简化方式，即不使用 i = i+1，而简写为 i++：

```
for (int i = 0; i < 100; i++)
{
  // ...
}
```

也可以在上面的例子中给循环变量 i 使用类型推断。使用类型推断时，循环结构变成：

```
for (var i = 0; i < 100; i++)
{
  // ...
}
```

嵌套的 for 循环非常常见，在每次迭代外部循环时，内部循环都要彻底执行完毕。这种模式通常用于在矩形多维数组中遍历每个元素。最外部的循环遍历每一行，内部的循环遍历某行上的每个列。下面的代码显示数字行，它还使用另一个 Console 方法 Console.Write()，该方法的作用与 Console.WriteLine()相同，但不在输出中添加回车换行符(代码文件 ForLoop/Program.cs )：

```
using System;

namespace Wrox
{
  class Program
  {
    static void Main()
    {
      // This loop iterates through rows
      for (int i = 0; i < 100; i+=10)
      {
        // This loop iterates through columns
        for (int j = i; j < i + 10; j++)
        {
          Console.Write($" {j}");
        }
        Console.WriteLine();
      }
    }
  }
}
```

尽管 j 是一个整数，但它会自动转换为字符串，以便进行连接。

上述例子的结果是：

```
0 1 2 3 4 5 6 7 8 9
10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47 48 49
50 51 52 53 54 55 56 57 58 59
60 61 62 63 64 65 66 67 68 69
70 71 72 73 74 75 76 77 78 79
```

```
80 81 82 83 84 85 86 87 88 89
90 91 92 93 94 95 96 97 98 99
```

尽管在技术上，可以在 for 循环的测试条件中计算其他变量，而不计算计数器变量，但这不太常见。也可以在 for 循环中忽略一个表达式(甚至所有表达式)。但此时，要考虑使用 while 循环。

### 2. while 循环

与 for 循环一样，while 循环也是一个预测试循环。其语法是类似的，但 while 循环只有一个表达式：

```
while(condition)
  statement(s);
```

与 for 循环不同的是，while 循环最常用于以下情况：在循环开始前，不知道重复执行一条语句或语句块的次数。通常，在某次迭代中，while 循环体中的语句把布尔标志设置为 false，结束循环，如下面的例子所示：

```
bool condition = false;
while (!condition)
{
  // This loop spins until the condition is true.
  DoSomeWork();
  condition = CheckCondition(); // assume CheckCondition() returns a bool
}
```

### 3. do…while 循环

do...while 循环是 while 循环的后测试版本。这意味着该循环的测试条件要在执行完循环体之后评估。因此 do...while 循环适用于循环体至少执行一次的情况：

```
bool condition;
do
{
  // This loop will at least execute once, even if Condition is false.
  MustBeCalledAtLeastOnce();
  condition = CheckCondition();
} while (condition);
```

### 4. foreach 循环

foreach 循环可以迭代集合中的每一项。现在不必考虑集合的准确概念(第 10 章将详细介绍集合)，只需要知道集合是一种包含一系列对象的对象即可。从技术上看，要使用集合对象，就必须支持 IEnumerable 接口。集合的例子有 C#数组、System.Collection 名称空间中的集合类，以及用户定义的集合类。从下面的代码中可以了解 foreach 循环的语法，其中假定 arrayOfInts 是一个 int 类型的数组：

```
foreach (int temp in arrayOfInts)
{
  Console.WriteLine(temp);
}
```

其中，foreach 循环每次迭代数组中的一个元素。它把每个元素的值放在 int 类型的变量 temp 中，然后执行一次循环迭代。

这里也可以使用类型推断。此时，foreach 循环变成：

```
foreach (var temp in arrayOfInts)
{
  // ...
}
```

temp 的类型推断为 int，因为这是集合项的类型。

注意，foreach 循环不能改变集合中各项(上面的 temp)的值，所以下面的代码不会编译：

```
foreach (int temp in arrayOfInts)
{
  temp++;
  Console.WriteLine(temp);
}
```

如果需要迭代集合中的各项，并改变它们的值，应使用 for 循环。

### 2.4.3　跳转语句

C#提供了许多可以立即跳转到程序中另一行代码的语句，在此，先介绍 goto 语句。

#### 1. goto 语句

goto 语句可以直接跳转到程序中用标签指定的另一行(标签是一个标识符，后跟一个冒号)：

```
goto Label1;
Console.WriteLine("This won't be executed");
Label1:
  Console.WriteLine("Continuing execution from here");
```

goto 语句有两个限制。不能跳转到像 for 循环这样的代码块中，也不能跳出类的范围；不能退出 try...catch 块后面的 finally 块(第 14 章将介绍如何用 try...catch...finally 块处理异常)。

在大多数情况下不允许使用 goto 语句。一般情况下，使用它肯定不是面向对象编程的好方式。

#### 2. break 语句

前面简要提到过 break 语句——在 switch 语句中使用它退出某个 case 语句。实际上，break 语句也可以用于退出 for、foreach、while 或 do...while 循环，该语句会使控制流执行循环后面的语句。

如果该语句放在嵌套的循环中，就执行最内部循环后面的语句。如果 break 放在 switch 语句或循环外部，就会产生编译错误。

#### 3. continue 语句

continue 语句类似于 break 语句，也必须在 for、foreach、while 或 do...while 循环中使用。但它只退出循环的当前迭代，开始执行循环的下一次迭代，而不是退出循环。

#### 4. return 语句

return 语句用于退出类的方法，把控制权返回给方法的调用者。如果方法有返回类型，return 语句必须返回这个类型的值；如果方法返回 void，应使用没有表达式的 return 语句。

## 2.5　名称空间

如前所述，名称空间提供了一种组织相关类和其他类型的方式。与文件或组件不同，名称空间是一种逻辑组合，而不是物理组合。在 C#文件中定义类时，可以把它包括在名称空间定义中。以后，在定义另一个类(在另一个文件中执行相关操作)时，就可以在同一个名称空间中包含它，创建一个逻辑组合，该组合告诉使用类的其他开发人员：这两个类是如何相关的以及如何使用它们：

```
using System;
namespace CustomerPhoneBookApp
{
  public struct Subscriber
  {
    // Code for struct here..
  }
}
```

把一个类型放在名称空间中，可以有效地给这个类型指定一个较长的名称，该名称包括类型的名称空间，名称之间用句点(.)隔开，最后是类名。在上面的例子中，Subscriber 结构的全名是 CustomerPhoneBookApp.Subscriber。这样，有相同短名的不同类就可以在同一个程序中使用了。全名常常称为完全限定的名称。

也可以在名称空间中嵌套其他名称空间，为类型创建层次结构：

```
namespace Wrox
{
  namespace ProCSharp
  {
```

```
      namespace Basics
      {
        class NamespaceExample
        {
          // Code for the class here..
        }
      }
    }
  }
```

每个名称空间名都由它所在名称空间的名称组成，这些名称用句点分隔开，开头是最外层的名称空间，最后是它自己的短名。所以 ProCSharp 名称空间的全名是 Wrox.ProCSharp，NamespaceExample 类的全名是 Wrox.ProCSharp.Basics.NamespaceExample。

使用这个语法也可以在自己的名称空间定义中组织名称空间，所以上面的代码也可以写为：

```
namespace Wrox.ProCSharp.Basics
{
  class NamespaceExample
  {
    // Code for the class here..
  }
}
```

注意不允许声明嵌套在另一个名称空间中的多部分名称空间。

名称空间与程序集无关。同一个程序集中可以有不同的名称空间，也可以在不同的程序集中定义同一个名称空间中的类型。

应在开始一个项目之前就计划定义名称空间的层次结构。一般可接受的格式是 CompanyName.ProjectName.SystemSection。所以在上面的例子中，Wrox 是公司名，ProCSharp 是项目，对于本章，Basics 是部分名。

### 2.5.1 using 语句

显然，名称空间相当长，输入起来很麻烦，用这种方式指定某个类也不总是必要的。如本章开头所述，C#允许简写类的全名。为此，要在文件的顶部列出类的名称空间，前面加上 using 关键字。在文件的其他地方，就可以使用其类型名称来引用名称空间中的类型了：

```
using System;
using Wrox.ProCSharp;
```

如前所述，很多 C#文件都以语句 using System;开头，这仅是因为微软公司提供的许多有用的类都包含在 System 名称空间中。

如果 using 语句引用的两个名称空间包含同名的类型，就必须使用完整的名称(或者至少较长的名称)，确保编译器知道访问哪个类型。例如，假如类 NamespaceExample 同时存在于 Wrox.ProCSharp.Basics 和 Wrox.ProCSharp.OOP 名称空间中。如果要在名称空间 Wrox.ProCSharp 中创建一个类 Test，并在该类中实例化一个 NamespaceExample 类，就需要指定使用哪个类：

```
using Wrox.ProCSharp.OOP;
using Wrox.ProCSharp.Basics;
namespace Wrox.ProCSharp
{
  class Test
  {
    static void Main()
    {
      Basics.NamespaceExample nSEx = new Basics.NamespaceExample();
      // do something with the nSEx variable.
    }
  }
}
```

公司应花一些时间开发一种名称空间模式，这样开发人员才能快速定位他们需要的功能，而且公司内部使用的类名也不会与现有的类库相冲突。本章后面将介绍建立名称空间模式的规则和其他命名约定。

### 2.5.2　名称空间的别名

using 关键字的另一个用途是给类和名称空间指定别名。如果名称空间的名称非常长，又要在代码中多次引用，但不希望该名称空间的名称包含在 using 语句中(例如，避免类名冲突)，就可以给该名称空间指定一个别名，其语法如下：

```
using alias = NamespaceName;
```

下面的例子(前面例子的修订版本)给 Wrox.ProCSharp.Basics 名称空间指定别名 Introduction，并使用这个别名实例化了在该名称空间中定义的 NamespaceExample 对象。注意名称空间别名的修饰符是“::”。因此将强制先从 Introduction 名称空间别名开始搜索。如果在相同的作用域中引入了 Introduction 类，就会发生冲突。即使出现了冲突，“::”运算符也允许引用别名。NamespaceExample 类有一个方法 GetNamespace()，该方法调用每个类都有的 GetType()方法，以访问表示类的类型的 Type 对象。下面使用这个对象返回类的名称空间名(代码文件 NamespaceSample/Program.cs )：

```
using System;
using Introduction = Wrox.ProCSharp.Basics;

class Program
{
  static void Main()
  {
    Introduction::NamespaceExample NSEx = new Introduction::NamespaceExample();
    Console.WriteLine(NSEx.GetNamespace());
  }
}

namespace Wrox.ProCSharp.Basics
{
  class NamespaceExample
  {
    public string GetNamespace()
    {
      return this.GetType().Namespace;
    }
  }
}
```

## 2.6　Main()方法

本章的开头提到过，C#程序是从方法 Main()开始执行的。根据执行环境，有不同的要求：

- 使用了 static 修饰符
- 在有任何名称的类中
- 返回 int 或 void 类型

虽然显式指定 public 修饰符是很常见的，因为按照定义，必须在程序外部调用该方法，但给该入口点方法指定什么访问级别并不重要，即使把该方法标记为 private，它也可以运行。

前面的例子只介绍了不带参数的 Main()方法。但在调用程序时，可以让 CLR 包含一个参数，将命令行参数传递给程序。这个参数是一个字符串数组，传统上称为args(但 C#可以接受任何名称)。在启动程序时，程序可以使用这个数组，访问通过命令行传送的选项。

下面的例子在传送给 Main()方法的字符串数组中循环，并把每个选项的值写入控制台窗口(代码文件 ArgumentsSample/Program.cs )：

```
using System;
namespace Wrox
{
  class Program
  {
    static void Main(string[] args)
    {
      for (int i = 0; i < args.Length; i++)
```

```
            {
              Console.WriteLine(args[i]);
            }
          }
        }
      }
```

在 Visual Studio 2017 中运行应用程序时，要给程序传递参数，可以在项目属性的 Debug 部分定义参数，如图 2-1 所示。运行应用程序，会在控制台上显示所有参数值。

使用.NET Core CLI 工具从命令行上运行应用程序时，只需要在 dotnet run 命令后面提供参数：

```
dotnet run arg1 arg2 arg3
```

如果想提供与 dotnet run 命令的参数相冲突的参数，可以在提供程序的参数之前添加两个短横号(-)：

```
dotnet run -- arg1 arg2 arg3
```

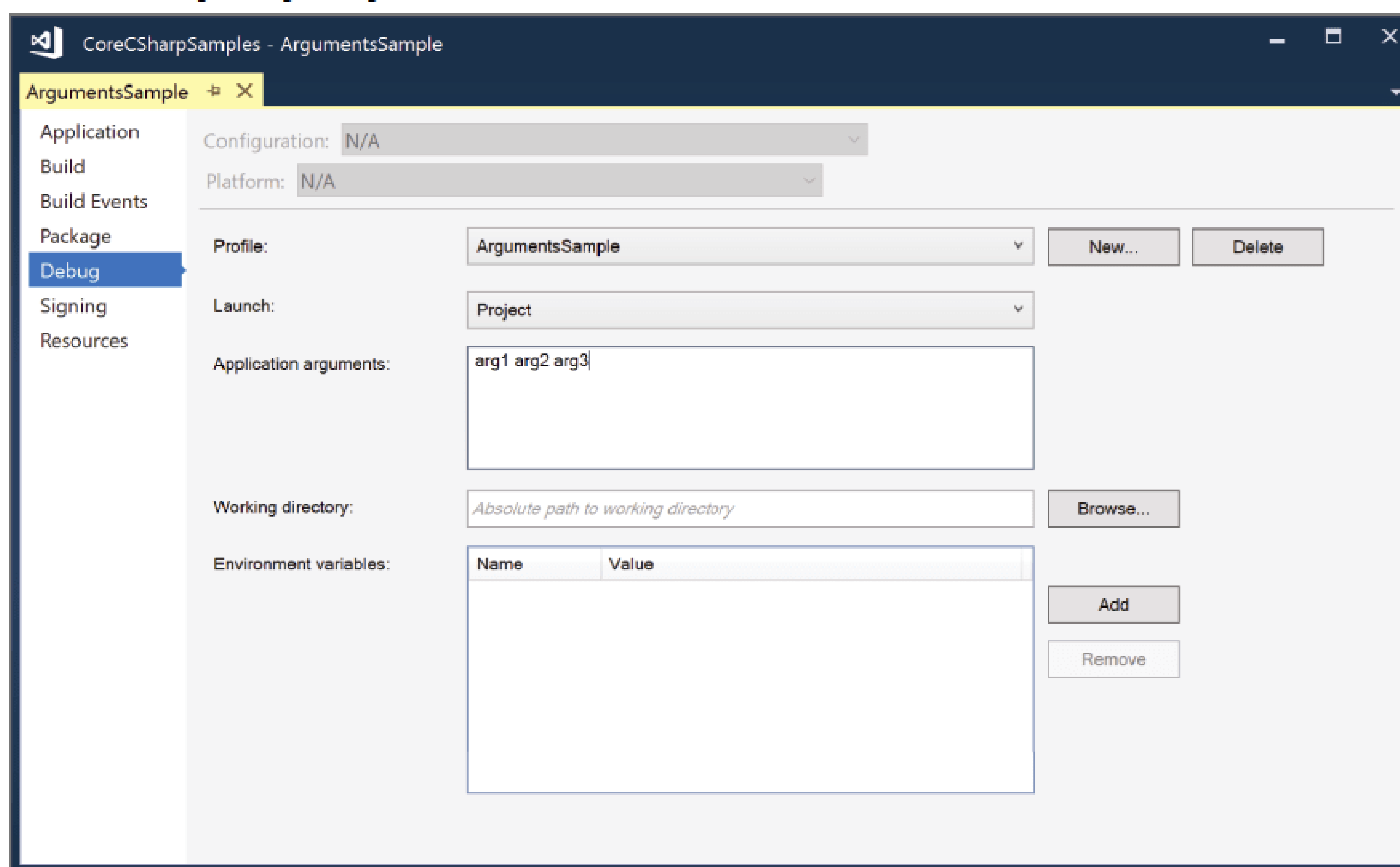



图 2-1

## 2.7 使用注释

本节的内容是给代码添加注释，该主题表面看来十分简单，但实际可能很复杂。注释有助于阅读代码的其他开发人员理解代码，而且可以用来为其他开发人员生成代码的文档。

### 2.7.1 源文件中的内部注释

本章开头提到过，C#使用传统的 C 风格注释方式：单行注释使用(// ...)，多行注释使用(/* ... */)：

```
// This is a single-line comment
/* This comment
spans multiple lines. */
```

单行注释中的任何内容，即从//开始一直到行尾的内容都会被编译器忽略。多行注释中“/*”和“*/”之间的所有内容也会被忽略。显然不能在多行注释中包含“*/”组合，因为这会被当成注释的结尾。

实际上，可以把多行注释放在一行代码中：

```
Console.WriteLine(/* Here's a comment! */ "This will compile.");
```

像这样的内联注释在使用时应小心，因为它们会使代码难以理解。但这样的注释在调试时是非常有用的，例如，在运行代码时要临时使用另一个值：

```
DoSomething(Width, /*Height*/ 100);
```

当然，字符串字面值中的注释字符会按照一般的字符来处理：

```
string s = "/* This is just a normal string .*/";
```

## 2.7.2　XML 文档

如前所述，除了 C 风格的注释外，C#还有一个非常出色的功能(本章将讨论这一功能)：根据特定的注释自动创建 XML 格式的文档说明。这些注释都是单行注释，但都以 3 条斜杠(///)开头，而不是通常的两条斜杠。在这些注释中，可以把包含类型和类型成员的文档说明的 XML 标记放在代码中。

编译器可以识别表 2-9 所示的标记。

表 2-9

| 标　　记 | 说　　明 |
|---|---|
| <c> | 把行中的文本标记为代码，例如<c>int i = 10;</c> |
| <code> | 把多行标记为代码 |
| <example> | 标记为一个代码示例 |
| <exception> | 说明一个异常类(编译器要验证其语法) |
| <include> | 包含其他文档说明文件的注释(编译器要验证其语法) |
| <list> | 把列表插入文档中 |
| <para> | 建立文本的结构 |
| <param> | 标记方法的参数(编译器要验证其语法) |
| <paramref> | 表明一个单词是方法的参数(编译器要验证其语法) |
| <permission> | 说明对成员的访问(编译器要验证其语法) |
| <remarks> | 给成员添加描述 |
| <returns> | 说明方法的返回值 |
| <see> | 提供对另一个参数的交叉引用(编译器要验证其语法) |
| <seealso> | 提供描述中的“参见”部分(编译器要验证其语法) |
| <summary> | 提供类型或成员的简短小结 |
| <typeparam> | 用在泛型类型的注释中，以说明一个类型参数 |
| <typeparamref> | 类型参数的名称 |
| <value> | 描述属性 |

要了解它们的工作方式，可以在 Calculator.cs 文件中添加一些 XML 注释。我们给类及其 Add()方法添加一个<summary>元素，也给 Add()方法添加一个<returns>元素和两个<param>元素：

```
// MathLib.cs
namespace Wrox.MathLib
{
  ///<summary>
  /// Wrox.MathLib.Calculator class.
  /// Provides a method to add two doublies.
  ///</summary>
  public class Calculator
  {
    ///<summary>
    /// The Add method allows us to add two doubles.
    ///</summary>
    ///<returns>Result of the addition (double)</returns>
    ///<param name="x">First number to add</param>
    ///<param name="y">Second number to add</param>
    public static double Add(double x, double y) => x + y;
  }
}
```

## 2.8 C#预处理器指令

除了前面介绍的常用关键字外，C#还有许多名为“预处理器指令”的命令。这些命令从来不会转换为可执行代码中的命令，但会影响编译过程的各个方面。例如，使用预处理器指令可以禁止编译器编译代码的某一部分。如果计划发布两个版本的代码，即基本版本和拥有更多功能的企业版本，就可以使用这些预处理器指令。在编译软件的基本版本时，使用预处理器指令可以禁止编译器编译与附加功能相关的代码。另外，在编写提供调试信息的代码时，也可以使用预处理器指令。实际上，在销售软件时，一般不希望编译这部分代码。

预处理器指令的开头都有符号#。

**注意：**

C++开发人员应该知道，在 C 和 C++中预处理器指令非常重要。但是，在 C#中，并没有那么多的预处理器指令，它们的使用也不太频繁。C#提供了其他机制来实现许多 C++指令的功能，如定制特性。还要注意，C#并没有一个像 C++那样的独立预处理器，所谓的预处理器指令实际上是由编译器处理的。尽管如此，C#仍保留了一些预处理器指令名称，因为这些命令会让人觉得就是预处理器。

下面简要介绍预处理器指令的功能。

### 2.8.1 #define 和#undef

#define 的用法如下所示：

```
#define DEBUG
```

它告诉编译器存在给定名称的符号，在本例中是 DEBUG。这有点类似于声明一个变量，但这个变量并没有真正的值，只是存在而已。这个符号不是实际代码的一部分，而只在编译器编译代码时存在。在 C#代码中它没有任何意义。

#undef 正好相反——它删除符号的定义：

```
#undef DEBUG
```

如果符号不存在，#undef 就没有任何作用。同样，如果符号已经存在，则#define 也不起作用。

必须把#define 和#undef 命令放在 C#源文件的开头位置，在声明要编译的任何对象的代码之前。

#define 本身并没有什么用，但与其他预处理器指令(特别是#if)结合使用时，它的功能就非常强大了。

**注意：**

这里应注意一般 C#语法的一些变化。预处理器指令不用分号结束，一般一行上只有一条命令。这是因为对于预处理器指令，C#不再要求命令使用分号进行分隔。如果编译器遇到一条预处理器指令，就会假定下一条命令在下一行。

### 2.8.2 #if、#elif、#else 和#endif

这些指令告诉编译器是否要编译代码块。考虑下面的方法：

```
int DoSomeWork(double x)
{
  // do something
  #if DEBUG
  Console.WriteLine($"x is {x}");
  #endif
}
```

这段代码会像往常那样编译，但 Console.WriteLine 方法调用包含在#if 子句内。这行代码只有在前面的#define 指令定义了符号 DEBUG 后才执行。当编译器遇到#if 指令后，将先检查相关的符号是否存在，如果符号存在，就编译#if 子句中的代码。否则，编译器会忽略所有的代码，直到遇到匹配的#endif 指令为止。一般是

在调试时定义符号 DEBUG，把与调试相关的代码放在#if 子句中。完成调试后，就把#define 指令注释掉，所有的调试代码会奇迹般地消失，可执行文件也会变小，最终用户不会被这些调试信息弄糊涂(显然，要做更多的测试，确保代码在没有定义 DEBUG 的情况下也能工作)。这项技术在 C 和 C++编程中十分常见，称为条件编译(conditional compilation)。

#elif (=else if)和#else 指令可以用在#if 块中，其含义非常直观。也可以嵌套#if 块：

```
#define ENTERPRISE
#define W10
// further on in the file
#if ENTERPRISE
// do something
#if W10
// some code that is only relevant to enterprise
// edition running on W10
#endif
#elif PROFESSIONAL
// do something else
#else
// code for the leaner version
#endif
```

#if 和#elif 还支持一组逻辑运算符“!”、“==”、“!=”和“||”。如果符号存在，就被认为是 true，否则为 false，例如：

```
#if W10 && (ENTERPRISE==false) // if W10 is defined but ENTERPRISE isn't
```

## 2.8.3 #warning 和 #error

另两个非常有用的预处理器指令是#warning 和#error。当编译器遇到它们时，会分别产生警告或错误。如果编译器遇到#warning 指令，会向用户显示#warning 指令后面的文本，之后编译继续进行。如果编译器遇到#error 指令，就会向用户显示后面的文本，作为一条编译错误消息，然后会立即退出编译，不会生成 IL 代码。

使用这些指令可以检查#define 语句是不是做错了什么事，使用#warning 语句可以提醒自己执行某个操作：

```
#if DEBUG && RELEASE
#error "You've defined DEBUG and RELEASE simultaneously!"
#endif
#warning "Don't forget to remove this line before the boss tests the code!"
Console.WriteLine("*I love this job.*");
```

## 2.8.4 #region 和#endregion

#region 和#endregion 指令用于把一段代码视为有给定名称的一个块，如下所示：

```
#region Member Field Declarations
int x;
double d;
Currency balance;
#endregion
```

这看起来似乎没有什么用，它根本不影响编译过程。这些指令真正的优点是它们可以被某些编辑器识别，包括 Visual Studio 编辑器。这些编辑器可以使用这些指令使代码在屏幕上更好地布局。第 18 章会详细介绍。

## 2.8.5 #line

#line 指令可以用于改变编译器在警告和错误信息中显示的文件名和行号信息。这条指令用得并不多。如果编写代码时，在把代码发送给编译器前，要使用某些软件包改变输入的代码，该指令最有用，因为这意味着编译器报告的行号或文件名与文件中的行号或编辑的文件名不匹配。#line 指令可以用于还原这种匹配。也可以使用语法#line default 把行号还原为默认的行号：

```
#line 164 "Core.cs" // We happen to know this is line 164 in the file
// Core.cs, before the intermediate
// package mangles it.
// later on
#line default // restores default line numbering
```

### 2.8.6 #pragma

#pragma 指令可以抑制或还原指定的编译警告。与命令行选项不同，#pragma 指令可以在类或方法级别实现，对抑制警告的内容和抑制的时间进行更精细的控制。下面的例子禁止“字段未使用”警告，然后在编译 MyClass 类后还原该警告：

```
#pragma warning disable 169
public class MyClass
{
  int neverUsedField;
}
#pragma warning restore 169
```

## 2.9 C#编程准则

本章的最后一节介绍编写 C#程序时应该牢记和遵循的准则。大多数 C#开发人员都遵守这些规则，所以在这些规则的指导下编写程序，可以方便其他开发人员使用程序的代码。

### 2.9.1 关于标识符的规则

本小节将讨论可用的变量、类和方法等的命名规则。注意本节所介绍的规则不仅是准则，也是 C#编译器强制使用的。

标识符是给变量、用户定义的类型(如类和结构)和这些类型的成员指定的名称。标识符区分大小写，所以 interestRate 和 InterestRate 是不同的变量。确定在 C#中可以使用什么标识符有两条规则：

- 尽管可以包含数字字符，但它们必须以字母或下划线开头。
- 不能把 C#关键字用作标识符。

C#包含如表 2-10 所示的保留关键字。

表 2-10

| | | | |
|---|---|---|---|
| abstract | event | new | struct |
| as | explicit | null | switch |
| base | extern | object | this |
| bool | false | operator | throw |
| break | finally | out | true |
| byte | fixed | override | try |
| case | float | params | typeof |
| catch | for | private | uint |
| char | foreach | protected | ulong |
| checked | goto | public | unchecked |
| class | if | readonly | unsafe |
| const | implicit | ref | ushort |
| continue | in | return | using |
| decimal | int | sbyte | virtual |
| default | interface | sealed | void |
| delegate | internal | short | volatile |
| do | is | sizeof | while |
| double | lock | stackalloc | |
| else | long | static | |
| enum | namespace | string | |

如果需要把某一保留字用作标识符(例如，访问一个用另一种语言编写的类)，那么可以在标识符的前面加上前缀符号@，告知编译器其后的内容是一个标识符，而不是 C#关键字(所以 abstract 不是有效的标识符，@abstract 才是)。

最后，标识符也可以包含 Unicode 字符，用语法\uXXXX 指定，其中 XXXX 是 Unicode 字符的 4 位十六进制编码。下面是有效标识符的一些例子：

- Name
- überfluß
- _Identifier
- \u005fIdentifier

最后两个标识符完全相同，可以互换(因为 005f 是下划线字符的 Unicode 代码)，所以这些标识符在同一个作用域内不要声明两次。注意，虽然从语法上看，在标识符中可以使用下划线字符，但大多数情况下最好不要这么做，因为它不符合微软公司的变量命名规则。这种命名规则可以确保开发人员使用相同的命名约定，易于阅读他人编写的代码。

**注意：**

*为什么 C#最新版本添加的一些新关键字没有列在保留字列表中？原因是，如果将它们添加到保留字列表中，就会破坏利用新 C#关键字的现有代码。解决方案是把这些关键字定义为上下文关键字，以改进语法；它们只能用在某些具体的代码中。例如，async 关键字只能用于方法声明，也可以用作变量名。编译器不会因此出现冲突。*

### 2.9.2　用法约定

在任何开发语言中，通常有一些传统的编程风格。这些风格不是语言自身的一部分，而是约定，例如，变量如何命名，类、方法或函数如何使用等。如果使用某语言的大多数开发人员都遵循相同的约定，不同的开发人员就很容易理解彼此的代码，这一般有助于程序的维护。约定主要取决于语言和环境。例如，在 Windows 平台上编程的 C++开发人员一般使用前缀 psz 或 lpsz 表示字符串：char *pszResult; char *lpszMessage;，但在 UNIX 系统上，则不使用任何前缀：char*Result; char *Message;。

从本书中的示例代码中可以总结出，C#中的约定是命名变量时不使用任何前缀：string Result; string Message;。

**注意：**

*变量名用带有前缀字母的方法来表示某种数据类型，这种约定称为 Hungarian 表示法。这样，其他阅读该代码的开发人员就可以立即从变量名中了解它代表什么数据类型。有了智能编辑器和 IntelliSense 之后，人们普遍认为 Hungarian 表示法是多余的。*

在许多语言中，用法约定是随着语言的使用逐渐演变而来的，但是对于 C#和整个.NET Framework，微软公司编写了非常多的用法准则，详见.NET/C# MSDN 文档。这说明，从一开始，.NET 程序就有非常高的互操作性，开发人员可以以此来理解代码。用法准则还得益于 20 年来面向对象编程的发展，因此相关的新闻组已经仔细考虑了这些用法规则，而且已经为开发团体所接受。所以我们应遵守这些准则。

但要注意，这些准则与语言规范不同。用户应尽可能遵循这些准则。但如果有很好的理由不遵循它们，也不会有什么问题。例如，不遵循这些准则，也不会出现编译错误。一般情况下，如果不遵循用法准则，就必须有充分的理由。准则应是正确的决策，而不是一种束缚。如果比较本书的后续示例，应注意到许多示例都没有遵循该约定。这通常是因为某些准则适用于大型程序，而不适合用于小示例。如果编写一个完整的软件包，就应遵循这些准则，但它们并不适合于只有 20 行代码的独立程序。在许多情况下，遵循约定会使这些示例难以理解。

编程风格的准则非常多。这里只介绍一些比较重要的，以及最适合于用户的准则。如果用户要让代码完全遵循用法准则，就需要参考 MSDN 文档。

1. **命名约定**

使程序易于理解的一个重要方面是给对象选择命名的方式，包括变量、方法、类、枚举和名称空间的命名方式。

显然，这些名称应反映对象的目的，且不与其他名称冲突。在.NET Framework 中，一般规则也是变量名要反映变量实例的目的，而不反映数据类型。例如，height 就是一个比较好的变量名，而 integerValue 就不太好。但是，这种规则是一种理想状态，很难达到。在处理控件时，大多数情况下使用 confirmationDialog 和 chooseEmployeeListBox 等变量名比较好，这些变量名说明了变量的数据类型。

名称的约定包括以下几个方面。

(1) 名称的大小写

在许多情况下，名称都应使用 Pascal 大小写形式。 Pascal 大小写形式指名称中单词的首字母大写，如 EmployeeSalary、ConfirmationDialog、PlainTextEncoding。注意，名称空间和类，以及基类中的成员等名称都应遵循 Pascal 大小写规则，最好不要使用带有下划线字符的单词，即名称不应是 employee_salary。其他语言中常量的名称常常全部大写，但在 C#中最好不要这样，因为这种名称很难阅读，而应全部使用 Pascal 大小写形式的命名约定：

```
const int MaximumLength;
```

还推荐使用另一种大小写模式：camel 大小写形式。这种形式类似于 Pascal 大小写形式，但名称中第一个单词的首字母不大写，如 employeeSalary、confirmationDialog、plainTextEncoding。有 3 种情况可以使用 camel 大小写形式：

- 类型中所有私有成员字段的名称：
  但要注意，成员字段的前缀名常常用一条下划线开头
- 传递给方法的所有参数的名称
- 用于区分同名的两个对象——比较常见的是属性封装字段：

```
private string employeeName;
public string EmployeeName
{
  get
  {
    return employeeName;
  }
}
```

如果这么做，则私有成员总是使用 camel 大小写形式，而公有的或受保护的成员总是使用 Pascal 大小写形式，这样使用这段代码的其他类就只能使用 Pascal 大小写形式的名称了(除了参数名以外)。

还要注意大小写问题。C#区分大小写，所以在 C#中，仅大小写不同的名称在语法上是正确的，如上面的例子所示。但是，有时可能从 Visual Basic 应用程序中调用程序集，而 Visual Basic 不区分大小写，如果使用仅大小写不同的名称，就必须使这两个名称不能在程序集的外部访问(上例是可行的，因为仅私有变量使用了 camel 大小写形式的名称)。否则，Visual Basic 中的其他代码就不能正确使用这个程序集。

(2) 名称的风格

名称的风格应保持一致。例如，如果类中的一个方法名为 ShowConfirmationDialog()，另一个方法就不能被命名为 ShowDialogWarning()或 WarningDialogShow()，而应是 ShowWarningDialog()。

(3) 名称空间的名称

名称空间的名称非常重要，一定要仔细考虑，以避免一个名称空间的名称与其他名称空间同名。记住，名称空间的名称是.NET 区分共享程序集中对象名的唯一方式。如果一个软件包的名称空间使用的名称与另一个软件包相同，而这两个软件包都由同一个程序使用，就会出问题。因此，最好用自己的公司名创建顶级的名称空间，再嵌套技术范围较窄、用户所在小组或部门或者类所在软件包的名称空间。Microsoft 建议使用名称空间：

<CompanyName>.<TechnologyName>，例如：

WeaponsOfDestructionCorp.RayGunControllers

WeaponsOfDestructionCorp.Viruses

(4) 名称和关键字

名称不应与任何关键字冲突，这非常重要。实际上，如果在代码中，试图给某一项指定与 C#关键字同名的名称，就会出现语法错误，因为编译器会假定该名称表示一条语句。但是，由于类可能由其他语言编写的代码访问，因此不能使用其他.NET 语言中的关键字作为对应的名称。一般来说，C++关键字类似于 C#关键字，不太可能与 C++混淆，只有 Visual C++常用的关键字以两个下划线字符开头。与 C#一样，C++关键字都是小写字母，如果要遵循公有类和成员使用 Pascal 风格名称的约定，则在它们的名称中至少有一个字母大写，因此不会与 C++关键字冲突。另一方面，Visual Basic 的问题会多一些，因为 Visual Basic 的关键字要比 C#的多，而且它不区分大小写，不能依赖于 Pascal 风格的名称来区分类和成员。

查看 MSDN 文档：https://docs.microsoft.com/dotnet/csharp/languagereference/keywords。在这里，有一个很长的 C#关键字列表，不应该用于类和成员。

### 2. 属性和方法的使用

类中出现混乱的一个方面是某个特定数量是用属性还是方法来表示。这没有硬性规定，但一般情况下，如果该对象的外观像变量，就应使用属性来表示它，即：

- 客户端代码应能读取它的值。最好不要使用只写属性，例如，应使用 SetPassword()方法，而不是 Password 只写属性。
- 读取该值不应花太长的时间。实际上，如果是属性，通常表明读取过程花的时间相对较短。
- 读取该值不应有任何明显的和不希望的负面效应。进一步说，设置属性的值，不应有与该属性不直接相关的负面效应。设置对话框的宽度会改变该对话框在屏幕上的外观，这是可以的，因为它与该属性相关。
- 可以按照任何顺序设置属性。尤其在设置属性时，最好不要因为还没有设置另一个相关的属性而抛出异常。例如，如果为了使用访问数据库的类，则需要设置 ConnectionString、UserName 和 Password，应确保已经实现了该类，这样用户才能按照任何顺序设置它们。
- 顺序读取属性应有相同的结果。如果属性的值可能会出现预料不到的改变，就应把它编写为一个方法。在监控汽车运动的类中，把 speed 设置为属性就不合适，而应使用 GetSpeed()方法；另一方面，应把 Weight 和 EngineSize 设置为属性，因为对于给定的对象，它们是不变的。

如果要编码的相关项满足上述所有条件，就把它设置为属性，否则就应使用方法。

### 3. 字段的使用

字段的用法非常简单。字段应总是私有的，但在某些情况下也可以把常量或只读字段设置为公有。原因是如果把字段设置为公有，就不利于在以后扩展或修改类。

遵循上面的准则就可以培养良好的编程习惯，而且这些准则应与面向对象的编程风格一起使用。

最后要记住以下有用的备注：微软公司在保持一致性方面相当谨慎，在编写.NET 基类时遵循了它自己的准则。在编写.NET 代码时应很好地遵循这些规则，对于基类来说，就是要弄清楚类、成员、名称空间的命名方式，以及类层次结构的工作方式等。类与基类之间的一致性有助于提高可读性和可维护性。

**注意：**

新的 ValueTuple 类型包含公共字段，而旧的 Tuple 类型则使用属性。微软打破了自己为字段定义的准则。由于元组的变量可以像 int 变量一样简单，性能非常重要，因此决定为值元组设置公共字段。它只是表明没有无例外的规则。有关元组的更多信息，请阅读第 13 章。

## 2.10　小结

本章介绍了一些 C#的基本语法，包括编写简单的 C#程序需要掌握的内容。我们讲述了许多基础知识，但其中有许多是熟悉 C 风格语言(甚至 JavaScript)的开发人员能立即领悟的。

C#语法与 C++/Java 语法非常类似，但仍存在一些细微区别。在许多领域，将这些语法与功能结合起来会提高编码速度，如高质量的字符串处理功能。C#还有一个已定义的强类型系统，该系统基于值类型和引用类型的区别。第 3 章和第 4 章将介绍 C#的面向对象编程特性。

# 第 3 章

# 对象和类型

**本章要点**

- 类和结构的区别
- 类成员
- 表达式体成员
- 按值和按引用传递参数
- 方法重载
- 构造函数和静态构造函数
- 只读字段
- 枚举
- 部分类
- 静态类
- Object 类，其他类型都从该类派生而来

**本章源代码下载地址(wrox.com)：**

打开 www.wrox.com 的 Download Code 选项卡可下载本章源代码。源代码也可以在 ObjectAndType 目录的 https://github.com/ProfessionalCSharp/ProfessionalCSharp7 中找到。

本章代码分为以下几个主要的示例文件：

- MathSample
- MethodSample
- StaticConstructorSample
- StructsSample
- PassingByValueAndByReference
- OutKeywordSample
- EnumSample
- ExtensionMethods

## 3.1 创建及使用类

到目前为止，我们介绍了组成 C#语言的主要模块，包括变量、数据类型和程序流语句，并简要介绍一个只包含 Main()方法的完整小例子。但还没有介绍如何把这些内容组合在一起，构成一个完整程序，其关键就在于对类的处理。这就是本章的主题。第 4 章将介绍继承以及与继承相关的特性。

**注意：**

本章将讨论与类相关的基本语法，但假定你已经熟悉了使用类的基本原则，例如，知道构造函数或属性的含义，因此本章主要阐述如何把这些原则应用于 C#代码。

## 3.2 类和结构

类和结构实际上都是创建对象的模板，每个对象都包含数据，并提供了处理和访问数据的方法。类定义了类的每个对象(称为实例)可以包含什么数据和功能。例如，如果一个类表示一个顾客，就可以定义字段 CustomerID、FirstName、LastName 和 Address，以包含该顾客的信息。还可以定义处理在这些字段中存储的数据的功能。接着，就可以实例化类的一个对象，来表示某个顾客，为这个实例设置相关字段的值，并使用其功能：

```
class PhoneCustomer
{
  public const string DayOfSendingBill = "Monday";
  public int CustomerID;
  public string FirstName;
  public string LastName;
}
```

结构不同于类，因为它们不需要在堆上分配空间(类是引用类型，总是存储在堆(heap)上)，而结构是值类型，通常存储在栈(stack)上，另外，结构不支持继承。

较小的数据类型使用结构可提高性能。在堆栈上存储值类型可以避免垃圾收集。结构的另一个用例与本机代码互操作；结构体的布局可以与本机数据类型相同。

但在语法上，结构与类非常相似，主要区别是使用关键字 struct 代替 class 来声明结构。例如，如果希望所有的 PhoneCustomer 实例都分布在栈上，而不是分布在托管堆上，就可以编写下面的语句：

```
struct PhoneCustomerStruct
{
  public const string DayOfSendingBill = "Monday";
  public int CustomerID;
  public string FirstName;
  public string LastName;
}
```

对于类和结构，都使用关键字 new 来声明实例：这个关键字创建对象并对其进行初始化。在下面的例子中，类和结构的字段值都默认为 0：

```
var myCustomer = new PhoneCustomer(); // works for a class
var myCustomer2 = new PhoneCustomerStruct();// works for a struct
```

在大多数情况下，类要比结构常用得多。因此，我们先讨论类，然后指出类和结构的区别，以及选择使用结构而不使用类的特殊原因。但除非特别说明，否则就可以假定用于类的代码也适用于结构。

**注意：**

类和结构的一个重要区别是，类类型的对象通过引用传递，结构类型的对象按值传递。详见本章后面的内容。

# 3.3　类

类包含成员，成员可以是静态成员或实例成员。静态成员属于类，实例成员属于对象。静态字段的值对每个对象都是相同的。而每个对象的实例字段都可以有不同的值。静态成员关联了 static 修饰符。成员的种类见表 3-1。

表 3-1

| 成　员 | 说　明 |
|---|---|
| 字段 | 字段是类的数据成员，它是类型的一个变量，该类型是类的一个成员 |
| 常量 | 常量与类相关(尽管它们没有 static 修饰符)。编译器使用真实值代替常量 |
| 方法 | 方法是与特定类相关联的函数 |
| 属性 | 属性是可以从客户端访问的函数组，其访问方式与访问类的公共字段类似。C#为读写类中的属性提供了专用语法，所以不必使用那些名称中嵌有 Get 或 Set 的方法。因为属性的这种语法不同于一般函数的语法，所以在客户端代码中，虚拟的对象被当作实际的东西 |
| 构造函数 | 构造函数是在实例化对象时自动调用的特殊函数。它们必须与所属的类同名，且不能有返回类型。构造函数用于初始化字段的值 |
| 索引器 | 索引器允许对象用访问数组的方式访问。索引器参见第 6 章 |
| 运算符 | 运算符执行的最简单的操作就是加法和减法。在两个整数相加时，严格地说，就是对整数使用“+”运算符。C#还允许指定把已有的运算符应用于自己的类(运算符重载)。第 6 章将详细论述运算符 |
| 事件 | 事件是类的成员，在发生某些行为(如修改类的字段或属性，或者进行了某种形式的用户交互操作)时，它可以让对象通知调用方。客户可以包含所谓“事件处理程序”的代码来响应该事件。第 8 章将详细介绍事件 |
| 析构函数 | 析构函数或终结器的语法类似于构造函数的语法，但是在 CLR 检测到不再需要某个对象时调用它。它们的名称与类相同，但前面有一个“~”符号。不可能预测什么时候调用终结器。终结器详见第 17 章 |
| 类型 | 类可以包含内部类。如果内部类型只和外部类型结合使用，就很有趣 |

下面详细介绍类成员。

## 3.3.1　字段

字段是与类相关的变量。前面的例子已经使用了 PhoneCustomer 类中的字段。

一旦实例化 PhoneCustomer 对象，就可以使用语法 Object.FieldName 来访问这些字段，如下例所示：

```
var customer1 = new PhoneCustomer();
customer1.FirstName = "Simon";
```

常量与类的关联方式和变量与类的关联方式相同。使用 const 关键字声明常量。如果把它声明为 public，就可以在类的外部访问它。

```
class PhoneCustomer
{
  public const string DayOfSendingBill = "Monday";
  public int CustomerID;
  public string FirstName;
  public string LastName;
}
```

## 3.3.2　只读字段

为了保证对象的字段不能改变，字段可以用 readonly 修饰符声明。带有 readonly 修饰符的字段只能在构造函数中分配值。它与 const 修饰符不同。编译器通过 const 修饰符，用其值取代了使用它的变量。编译器知道常量的值。只读字段在运行期间通过构造函数指定。与常量字段相反，只读字段可以是实例成员。使用只读字段

作为类成员时，需要把 static 修饰符分配给该字段。

如果有一个用于编辑文档的程序，因为要注册，所以需要限制可以同时打开的文档数。现在假定要销售该软件的不同版本，而且顾客可以升级他们的版本，以便同时打开更多的文档。显然，不能在源代码中对最大文档数进行硬编码，而是需要一个字段来表示这个最大文档数。这个字段必须是只读的——每次启动程序时，从某个文件存储中读取。代码如下所示：

```
public class DocumentEditor
{
  private static readonly uint s_maxDocuments;
  static DocumentEditor()
  {
    s_maxDocuments = DoSomethingToFindOutMaxNumber();
  }
}
```

在本例中，字段是静态的，因为每次运行程序的实例时，只需要存储最大文档数一次。这就是在静态构造函数中初始化它的原因。如果只读字段是一个实例字段，就要在实例构造函数中初始化它。例如，假定编辑的每个文档都有一个创建日期，但不允许用户修改它(因为这会覆盖过去的日期)。

如前所述，日期用基类 System.DateTime 表示。下面的代码在构造函数中使用 DateTime 结构初始化 _creationTime 字段。初始化 Document 类后，创建时间就不能改变了：

```
public class Document
{
  private readonly DateTime _creationTime;
  public Document()
  {
    _creationTime = DateTime.Now;
  }
}
```

在上面的代码段中，_creationTime 和 s_maxDocuments 的处理方式与任何其他字段相同，但因为它们是只读的，所以不能在构造函数外部赋值：

```
void SomeMethod()
{
  s_maxDocuments = 10; // compilation error here. MaxDocuments is readonly
}
```

还要注意，在构造函数中不必给只读字段赋值。如果没有赋值，它的值就是其特定数据类型的默认值，或者在声明时给它初始化的值。这适用于只读的静态字段和实例字段。

最好不把字段声明为 public。如果修改类的公共成员，使用这个公共成员的每个调用程序也需要更改。例如，如果希望在下一个版本中检查最大的字符串长度，公共字段就需要更改为一个属性。使用公共字段的现有代码，必须重新编译，才能使用这个属性(尽管在调用程序看来，语法与属性相同)。如果只在现有的属性中改变检查，那么调用程序不需要重新编译就能使用新版本。

最好把字段声明为 private，使用属性来访问字段，如下一节所述。

### 3.3.3 属性

属性(property)的概念是：它是一个方法或一对方法，在客户端代码看来，它(们)是一个字段。

下面把前面示例中变量名为_firstName 的名字字段改为私有。FirstName 属性包含 get 和 set 访问器，来检索和设置支持字段的值：

```
class PhoneCustomer
{
  private string _firstName;
  public string FirstName
  {
    get
    {
      return _firstName;
    }
    set
    {
```

```
      _firstName = value;
    }
  }
  //...
}
```

get 访问器不带任何参数，且必须返回属性声明的类型。也不应为 set 访问器指定任何显式参数，但编译器假定它带一个参数，其类型也与属性相同，并表示为 value。

下面的示例使用另一个命名约定。下面的代码包含一个属性 Age，它设置了一个字段 age。在这个例子中，age 表示属性 Age 的后备变量。

```
private int age;
public int Age
{
  get
  {
    return age;
  }
  set
  {
    age = value;
  }
}
```

注意这里所用的命名约定。我们采用 C#的区分大小写模式，使用相同的名称，但公有属性采用 Pascal 大小写形式命名，如果存在一个等价的私有字段，则它采用 camel 大小写形式命名。在早期.NET 版本中，此命名约定由微软的 C#团队优先使用。最近他们使用的命名约定是给字段名加上下划线作为前缀。这会为识别字段而不是局部变量提供极大的便利。

**注意：**

*微软团队针对不同情况使用不同的命名约定。在使用类型的私有成员时，.NET 没有严格的命名约定。然而，在团队里应该使用相同的约定。.NET Core 团队转向使用下划线作为字段的前缀，这是本书大多数地方使用的约定(参见 https://github.com/dotnet/ corefx/blob/master/Documentation/coding-guidelines/coding-style.md)。*

#### 1. 具有表达式体的属性访问器

使用 C# 7，还可以将属性访问器编写为具有表达式体的成员。例如，前面显示的属性 FirstName 可以使用=>编写。这个新特性减少了编写大括号的需求，并且使用 get 访问器省略了 return 关键字。

```
private string _firstName;
public string FirstName
{
  get => _firstName;
  set => _firstName = value;
}
```

使用具有表达式体的成员时，属性访问器的实现只能由一条语句组成。

#### 2. 自动实现的属性

如果属性的 set 和 get 访问器中没有任何逻辑，就可以使用自动实现的属性。这种属性会自动实现后备成员变量。前面 Age 示例的代码如下：

```
public int Age { get; set; }
```

不需要声明私有字段。编译器会自动创建它。使用自动实现的属性，就不能直接访问字段，因为不知道编译器生成的名称。如果对属性所需要做的就是读取和编写一个字段，那么使用自动实现属性时的属性语法比使用具有表达式体的属性访问器时的语法要短。

使用自动实现的属性，就不能在属性设置中验证属性的有效性。所以在上面的例子中，不能检查是否设置了无效的年龄。

自动实现的属性可以使用属性初始化器来初始化：

```
public int Age { get; set; } = 42;
```

### 3. 属性的访问修饰符

C#允许给属性的 get 和 set 访问器设置不同的访问修饰符，所以属性可以有公有的 get 访问器和私有或受保护的 set 访问器。这有助于控制属性的设置方式或时间。在下面的代码示例中，注意 set 访问器有一个私有访问修饰符，而 get 访问器没有任何访问修饰符。这表示 get 访问器具有属性的访问级别。在 get 和 set 访问器中，必须有一个具备属性的访问级别。如果 get 访问器的访问级别是 protected，就会产生一个编译错误，因为这会使两个访问器的访问级别都不是属性。

```
public string Name
{
  get => _name;
  private set => _name = value;
}
```

通过自动实现的属性，也可以设置不同的访问级别：

```
public int Age { get; private set; }
```

**注意：**

一些开发人员可能会担心，前面列举了许多情况，其中标准 C#编码方式导致了大材小用。例如，通过属性访问字段，而不是直接访问字段。这些额外的函数调用是否会增加系统开销，导致性能下降？其实，不需要担心这种编程方式会在 C#中带来性能损失。C#代码会编译为 IL，然后在运行时 JIT 编译为本地可执行代码。JIT 编译器可生成高度优化的代码，并在适当的时候随意地内联代码(即，用内联代码来替代函数调用)。如果实现某个方法或属性仅是调用另一个方法，或返回一个字段，则该方法或属性肯定是内联的。

通常不需要改变内联的行为，但在通知编译器有关内联的情况时有一些控制。使用属性 MethodImpl 可以定义不应用内联的方法(MethodImplOptions.NoInlining)，或内联应该由编译器主动完成(MethodImplOptions.AggressiveInlining)。对于属性，需要直接将这个属性应用于 get 和 set 访问器。属性详见第 16 章。

### 4. 只读属性

在属性定义中省略 set 访问器，就可以创建只读属性。因此，如下代码把 Name 变成只读属性：

```
private readonly string _name;
public string Name
{
  get => _name;
}
```

用 readonly 修饰符声明字段，只允许在构造函数中初始化属性的值。

**注意：**

可以创建只读属性，就可以创建只写属性。只要在属性定义中省略 get 访问器，就可以创建只写属性。但是，这是不好的编程方式，因为这可能会使客户端代码的作者感到迷惑。一般情况下，如果要这么做，最好使用一个方法替代。

### 5. 自动实现的只读属性

C# 提供了一个简单的语法，使用自动实现的属性创建只读属性，访问只读字段。这些属性可以使用属性初始化器来初始化。

```
public string Id { get; } = Guid.NewGuid().ToString();
```

在后台，编译器会创建一个只读字段和一个属性，其 get 访问器可以访问这个字段。初始化器的代码进入构造函数的实现代码，并在调用构造函数体之前调用。

当然，只读属性也可以显式地在构造函数中初始化，如下面的代码片段所示：

```
public class Person
{
  public Person(string name) => Name = name;
```

```
  public string Name { get; }
}
```

### 6. 表达式体属性

从 C# 6 开始，只有 get 访问器的属性可以使用表达式体属性实现。类似于表达式体方法，表达式体属性不需要花括号和返回语句。表达式体属性是带有 get 访问器的属性，但不需要编写 get 关键字。只是 get 访问器的实现后跟 lambda 操作符。对于 Person 类，FullName 属性使用表达式体属性实现，通过该属性返回 FirstName 和 LastName 属性值的组合(代码文件 ClassesSample/Program.cs)：

```
public class Person
{
  public Person(string firstName, string lastName)
  {
    FirstName = firstName;
    LastName = lastName;
  }
  public string FirstName { get; }
  public string LastName { get; }
  public string FullName => $"{FirstName} {LastName}";
}
```

### 7. 不可变的类型

如果类型包含可以改变的成员，它就是一个可变的类型。使用 readonly 修饰符，编译器会在状态改变时报错。状态只能在构造函数中初始化。如果对象没有任何可以改变的成员，只有只读成员，它就是一个不可变类型。其内容只能在初始化时设置。这对于多线程是非常有用的，因为多个线程可以访问信息永远不会改变的同一个对象。因为内容不能改变，所以不需要同步。

不可变类型的一个例子是 String 类。这个类没有定义任何允许改变其内容的成员。诸如 ToUpper(把字符串更改为大写)的方法总是返回一个新的字符串，但传递到构造函数的原始字符串保持不变。

**注意：**
.NET 也提供了不可变的集合，这些集合类参见第 11 章。

## 3.3.4　匿名类型

第 2 章讨论了 var 关键字，它用于表示隐式类型化的变量。var 与 new 关键字一起使用时，可以创建匿名类型。匿名类型只是一个继承自 Object 且没有名称的类。该类的定义从初始化器中推断，类似于隐式类型化的变量。

如果需要一个对象包含某个人的姓氏、中间名和名字，则声明如下：

```
var captain = new
{
  FirstName = "James",
  MiddleName = "T",
  LastName = "Kirk"
};
```

这会生成一个包含 FirstName、MiddleName 和 LastName 属性的对象。如果创建另一个对象，如下所示：

```
var doctor = new
{
  FirstName = "Leonard",
  MiddleName = string.Empty,
  LastName = "McCoy"
};
```

那么 captain 和 doctor 的类型就相同。例如，可以设置 captain = doctor。只有所有属性都匹配，才能设置 captain = doctor。

如果所设置的值来自于另一个对象，则可以推断匿名类型成员的名称。这样，就可以简化初始化器。如果已经有一个包含 FirstName、MiddleName 和 LastName 属性的类，且有该类的一个实例 person，captain 对象就可以初始化为：

```
var captain = new
{
  person.FirstName,
  person.MiddleName,
  person.LastName
};
```

person 对象的属性名应投射到新对象名 captain，所以 captain 对象应有 FirstName、MiddleName 和 LastName 属性。

这些新对象的类型名未知。编译器为类型“伪造”了一个名称，但只有编译器才能使用它。我们不能也不应使用新对象上的任何类型反射，因为这不会得到一致的结果。

### 3.3.5 方法

注意，正式的 C#术语区分函数和方法。在 C#术语中，“函数成员”不仅包含方法，也包含类或结构的一些非数据成员，如索引器、运算符、构造函数和析构函数等，甚至还有属性。这些都不是数据成员，字段、常量和事件才是数据成员。

#### 1. 方法的声明

在 C#中，方法的定义包括任意方法修饰符(如方法的可访问性)、返回值的类型，然后依次是方法名、输入参数的列表(用圆括号括起来)和方法体(用花括号括起来)。

```
[modifiers] return_type MethodName([parameters])
{
  // Method body
}
```

每个参数都包括参数的类型名和在方法体中的引用名称。但如果方法有返回值，则 return 语句就必须与返回值一起使用，以指定出口点，例如：

```
public bool IsSquare(Rectangle rect)
{
  return (rect.Height == rect.Width);
}
```

如果方法没有返回值，就把返回类型指定为 void，因为不能省略返回类型。如果方法不带参数，仍需要在方法名的后面包含一对空的圆括号()。此时 return 语句就是可选的——当到达闭花括号时，方法会自动返回。

#### 2. 表达式体方法

如果方法的实现只有一条语句，C#为方法定义提供了一个简化的语法：表达式体方法。使用新的语法，不需要编写花括号和 return 关键字，而使用运算符=>区分操作符左边的声明和操作符右边的实现代码。

下面的例子与前面的方法 IsSquare 相同，但使用表达式体方法语法实现。lambda 操作符的右侧定义了方法的实现代码。不需要花括号和返回语句。返回的是语句的结果，该结果的类型必须与左边方法声明的类型相同，在下面的代码片段中，该类型是 bool：

```
public bool IsSquare(Rectangle rect) => rect.Height == rect.Width;
```

#### 3. 调用方法

在下面的例子中，说明了类的定义和实例化、方法的定义和调用的语法。类 Math 定义了静态成员和实例成员(代码文件 MathSample/Math.cs )：

```
public class Math
{
  public int Value { get; set; }
  public int GetSquare() => Value * Value;
```

```
  public static int GetSquareOf(int x) => x * x;
  public static double GetPi() => 3.14159;
}
```

Program 类利用 Math 类调用静态方法并实例化一个对象，来调用实例成员(代码文件 MathSample/Program.cs)：

```
using System;
namespace MathSample
{
  class Program
  {
    static void Main()
    {
      // Try calling some static functions.
      Console.WriteLine($"Pi is {Math.GetPi()}");
      int x = Math.GetSquareOf(5);
      Console.WriteLine($"Square of 5 is {x}");
      // Instantiate a Math object
      var math = new Math(); // instantiate a reference type
      // Call instance members
      math.Value = 30;
      Console.WriteLine($"Value field of math variable contains {math.Value}");
      Console.WriteLine($"Square of 30 is {math.GetSquare()}");
    }
  }
}
```

运行 MathSample 示例，会得到如下结果：

```
Pi is 3.14159
Square of 5 is 25
Value field of math variable contains 30
Square of 30 is 900
```

从代码中可以看出，Math 类包含一个属性和一个方法，该属性包含一个数字，该方法计算该数字的平方。这个类还包含两个静态方法，一个返回 pi 的值，另一个计算作为参数传入的数字的平方。

这个类有一些功能并不是设计 C#程序的好例子。例如，GetPi()通常作为 const 字段来执行，而好的设计应使用目前还没有介绍的概念。

#### 4. 方法的重载

C#支持方法的重载——方法的几个版本有不同的签名(即，方法名相同，但参数的个数和/或数据类型不同)。为了重载方法，只需要声明同名但参数个数或类型不同的方法即可：

```
class ResultDisplayer
{
  public void DisplayResult(string result)
  {
    // implementation
  }

  public void DisplayResult(int result)
  {
    // implementation
  }
}
```

不仅参数类型可以不同，参数的数量也可以不同，如下一个示例所示。一个重载的方法可以调用另一个重载的方法：

```
class MyClass
{
  public int DoSomething(int x)
  {
    return DoSomething(x, 10); // invoke DoSomething with two parameters
  }

  public int DoSomething(int x, int y)
  {
    // implementation
  }
}
```

**注意：**

对于方法重载，仅通过返回类型不足以区分重载的版本。仅通过参数名称也不足以区分它们。需要区分参数的数量和/或类型。

### 5. 命名的参数

调用方法时，变量名不需要添加到调用中。然而，如果有如下的方法签名，用于移动矩形：

```
public void MoveAndResize(int x, int y, int width, int height)
```

用下面的代码片段调用它，就不能从调用中看出使用了什么数字，这些数字用于哪里：

```
r.MoveAndResize(30, 40, 20, 40);
```

可以改变调用，明确数字的含义：

```
r.MoveAndResize(x: 30, y: 40, width: 20, height: 40);
```

任何方法都可以使用命名的参数调用。只需要编写变量名，后跟一个冒号和所传递的值。编译器会去掉变量名，创建一个方法调用，就像没有变量名一样——这在编译后的代码中没有差别。C# 7.2 允许使用不拖尾的命名参数。使用早期的 C#版本时，需要在使用第一个命名参数之后为所有参数提供名称。

还可以用这种方式更改变量的顺序，编译器会重新安排，获得正确的顺序。其真正的优势是下一节所示的可选参数。

### 6. 可选参数

参数也可以是可选的。必须为可选参数提供默认值。可选参数还必须是方法定义的最后的参数：

```
public void TestMethod(int notOptionalNumber, int optionalNumber = 42)
{
  Console.WriteLine(optionalNumber + notOptionalNumber);
}
```

这个方法可以使用一个或两个参数调用。传递一个参数，编译器就修改方法调用，给第二个参数传递 42。

```
TestMethod(11);
TestMethod(11, 22);
```

**注意：**

因为编译器用可选参数改变方法，传递默认值，所以在程序集的新版本中，默认值不应该改变。在新版本中修改默认值，如果调用程序在没有重新编译的另一个程序集中，就会使用旧的默认值。这就是为什么应该只给可选参数提供永远不会改变的值。如果默认值更改时，总是重新编译调用的方法，这就不是一个问题。

可以定义多个可选参数，如下所示：

```
public void TestMethod(int n, int opt1 = 11, int opt2 = 22, int opt3 = 33)
{
  Console.WriteLine(n + opt1 + opt2 + opt3);
}
```

这样，该方法就可以使用 1、2、3 或 4 个参数调用。下面代码中的第一行给可选参数指定值 11、22 和 33。第二行传递了前三个参数，最后一个参数的值是 33：

```
TestMethod(1);
TestMethod(1, 2, 3);
```

通过多个可选参数，命名参数的特性就会发挥作用。使用命名参数，可以传递任何可选参数，例如，下面的例子仅传递最后一个参数：

```
TestMethod(1, opt3: 4);
opt3: 4
```

**注意：**

注意使用可选参数时的版本控制问题。一个问题是在新版本中改变默认值；另一个问题是改变参数的数量。添加另一个可选参数看起来很容易，因为它是可选的。然而，编译器更改调用代码，填充所有的参数，如果以后添加另一个参数，早期编译的调用程序就会失败。

#### 7. 个数可变的参数

使用可选参数，可以定义数量可变的参数。然而，还有另一种语法允许传递数量可变的参数——这个语法没有版本控制问题。

声明数组类型的参数(示例代码使用一个 int 数组)，添加 params 关键字，就可以使用任意数量的 int 参数调用该方法。

```
public void AnyNumberOfArguments(params int[] data)
{
  foreach (var x in data)
  {
    Console.WriteLine(x);
  }
}
```

**注意：**

数组参见第 7 章。

AnyNumberOfArguments 方法的参数类型是 int[]，可以传递一个 int 数组，或因为 params 关键字，可以传递一个或任何数量的 int 值：

```
AnyNumberOfArguments(1);
AnyNumberOfArguments(1, 3, 5, 7, 11, 13);
```

如果应该把不同类型的参数传递给方法，可以使用 object 数组：

```
public void AnyNumberOfArguments(params object[] data)
{
  // ...
```

现在可以使用任何类型调用这个方法：

```
AnyNumberOfArguments("text", 42);
```

如果 params 关键字与方法签名定义的多个参数一起使用，则 params 只能使用一次，而且它必须是最后一个参数：

```
Console.WriteLine(string format, params object[] arg);
```

前面介绍了方法的许多方面，下面看看构造函数，这是一种特殊的方法。

### 3.3.6　构造函数

声明基本构造函数的语法就是声明一个与包含的类同名的方法，但该方法没有返回类型：

```
public class MyClass
{
  public MyClass()
  {
  }

  // rest of class definition
}
```

没有必要给类提供构造函数，到目前为止本书的例子中没有提供这样的构造函数。一般情况下，如果没有提供任何构造函数，编译器会在后台生成一个默认的构造函数。这是一个非常基本的构造函数，它只能把所有的成员字段初始化为标准的默认值(例如，引用类型为空引用，数值数据类型为 0，bool 为 false)。这通常就足够了，否则就需要编写自己的构造函数。

构造函数的重载遵循与其他方法相同的规则。换言之，可以为构造函数提供任意多的重载，只要它们的签

名有明显的区别即可：

```
public MyClass() // zeroparameter constructor
{
  // construction code
}

public MyClass(int number) // another overload
{
  // construction code
}
```

但是，如果提供了带参数的构造函数，编译器就不会自动提供默认的构造函数。只有在没有定义任何构造函数时，编译器才会自动提供默认的构造函数。在下面的例子中，因为定义了一个带单个参数的构造函数，编译器会假定这是可用的唯一构造函数，所以它不会隐式地提供其他构造函数：

```
public class MyNumber
{
  private int _number;
  public MyNumber(int number)
  {
    _number = number;
  }
}
```

如果试图使用无参数的构造函数实例化 MyNumber 对象，就会得到一个编译错误：

```
var numb = new MyNumber(); // causes compilation error
```

注意，可以把构造函数定义为 private 或 protected，这样不相关的类就不能访问它们：

```
public class MyNumber
{
  private int _number;
  private MyNumber(int number) // another overload
  {
    _number = number;
  }
}
```

这个例子没有为 MyNumber 定义任何公有的或受保护的构造函数。这就使 MyNumber 不能使用 new 运算符在外部代码中实例化(但可以在 MyNumber 中编写一个公有静态属性或方法，以实例化该类)。这在下面两种情况下是有用的：

- 类仅用作某些静态成员或属性的容器，因此永远不会实例化它。在这种情况下，可以用 static 修饰符声明类。使用这个修饰符，类只能包含静态成员，不能实例化。
- 希望类仅通过调用某个静态成员函数来实例化(这就是所谓对象实例化的类工厂方法)。单例模式的实现如下面的代码片段所示：

```
public class Singleton
{
  private static Singleton s_instance;
  private int _state;
  private Singleton(int state)
  {
    _state = state;
  }
  public static Singleton Instance
  {
    get => s_instance ?? (s_instance = new Singleton(42);
  }
}
```

Singleton 类包含一个私有构造函数，所以只能在类中实例化它本身。为了实例化它，静态属性 Instance 返回字段 s_instance。如果这个字段尚未初始化(null)，就调用实例构造函数，创建一个新的实例。为了检查 null，使用合并运算符。如果这个操作符的左边是 null，就处理操作符的右边，调用实例构造函数。

**注意：**
合并运算符参见第 6 章。

### 1. 表达式体和构造函数

如果构造函数的实现由一个表达式组成，那么构造函数可以通过一个表达式体来实现：

```
public class Singleton
{
  private static Singleton s_instance;
  private int _state;
  private Singleton(int state) => _state = state;

  public static Singleton Instance =>
    s_instance ?? (s_instance = new Singleton(42);
}
```

### 2. 从构造函数中调用其他构造函数

有时，在一个类中有几个构造函数，以容纳某些可选参数，这些构造函数包含一些共同的代码。例如，下面的情况：

```
class Car
{
  private string _description;
  private uint _nWheels;

  public Car(string description, uint nWheels)
  {
    _description = description;
    _nWheels = nWheels;
  }

  public Car(string description)
  {
    _description = description;
    _nWheels = 4;
  }
  // ...
}
```

这两个构造函数初始化相同的字段，显然，最好把所有代码放在一个地方。C#有一个特殊的语法，称为构造函数初始化器，可以实现此目的：

```
class Car
{
  private string _description;
  private uint _nWheels;
  public Car(string description, uint nWheels)
  {
    _description = description;
    _nWheels = nWheels;
  }
  public Car(string description): this(description, 4)
  {
  }
  // ...
```

这里，this 关键字仅调用参数最匹配的那个构造函数。注意，构造函数初始化器在构造函数的函数体之前执行。现在假定运行下面的代码：

```
var myCar = new Car("Proton Persona");
```

在本例中，在带一个参数的构造函数的函数体执行之前，先执行带两个参数的构造函数(但在本例中，因为在带一个参数的构造函数的函数体中没有代码，所以没有区别)。

C#构造函数初始化器可以包含对同一个类的另一个构造函数的调用(使用前面介绍的语法)，也可以包含对直接基类的构造函数的调用(使用相同的语法，但应使用 base 关键字代替 this)。初始化器中不能有多个调用。

### 3. 静态构造函数

C#的一个特征是也可以给类编写无参数的静态构造函数。这种构造函数只执行一次，而前面的构造函数是实例构造函数，只要创建类的对象，就会执行它。

```
class MyClass
{
  static MyClass()
  {
    // initialization code
  }
  // rest of class definition
}
```

编写静态构造函数的一个原因是，类有一些静态字段或属性，需要在第一次使用类之前，从外部源中初始化这些静态字段和属性。

.NET 运行库没有确保什么时候执行静态构造函数，所以不应把要求在某个特定时刻(例如，加载程序集时)执行的代码放在静态构造函数中。也不能预计不同类的静态构造函数按照什么顺序执行。但是，可以确保静态构造函数至多运行一次，即在代码引用类之前调用它。在 C#中，通常在第一次调用类的任何成员之前执行静态构造函数。

注意，静态构造函数没有访问修饰符，其他 C#代码从来不显式调用它，但在加载类时，总是由.NET 运行库调用它，所以像 public 或 private 这样的访问修饰符就没有任何意义。出于同样的原因，静态构造函数不能带任何参数，一个类也只能有一个静态构造函数。很显然，静态构造函数只能访问类的静态成员，不能访问类的实例成员。

无参数的实例构造函数与静态构造函数可以在同一个类中定义。尽管参数列表相同，但这并不矛盾，因为在加载类时执行静态构造函数，而在创建实例时执行实例构造函数，所以何时执行哪个构造函数不会有冲突。

如果多个类都有静态构造函数，先执行哪个静态构造函数就不确定。此时静态构造函数中的代码不应依赖于其他静态构造函数的执行情况。另一方面，如果任何静态字段有默认值，就在调用静态构造函数之前分配它们。

下面用一个例子来说明静态构造函数的用法。该例子的思想基于包含用户首选项的程序(假定用户首选项存储在某个配置文件中)。为了简单起见，假定只有一个用户首选项——BackColor，它表示要在应用程序中使用的背景色。因为这里不想编写从外部数据源中读取数据的代码，所以假定该首选项在工作日的背景色是红色，在周末的背景色是绿色。程序仅在控制台窗口中显示首选项——但这足以说明静态构造函数是如何工作的。

类 UserPreferences 用 static 修饰符声明，因此它不能实例化，只能包含静态成员。静态构造函数根据星期几初始化 BackColor 属性 (代码文件 StaticConstructorSample /UserPreferences.cs)：

```
public static class UserPreferences
{
  public static Color BackColor { get; }
  static UserPreferences()
  {
    DateTime now = DateTime.Now;
    if (now.DayOfWeek == DayOfWeek.Saturday
        || now.DayOfWeek == DayOfWeek.Sunday)
    {
      BackColor = Color.Green;
    }
    else
    {
      BackColor = Color.Red;
    }
  }
}
```

这段代码使用了.NET Framework 类库提供的 System.DateTime 结构。DateTime 结构实现了返回当前时间的静态属性 Now，DayOfWeek 属性是 DateTime 的实例属性，返回一个类型 DayOfWeek 的枚举值。

Color 定义为 enum 类型，包含几种颜色。enum 类型详见“枚举”一节(代码文件 StaticConstructor-Sample/Enum.cs)：

```
public enum Color
{
  White,
  Red,
  Green,
  Blue,
```

```
  Black
}
```

Main()方法调用 Console.WriteLine 方法，把用户首选的背景色写到控制台(代码文件 StaticConstructorSample/Program.cs)：

```
class Program
{
  static void Main()
  {
    Console.WriteLine(
      $"User-preferences: BackColor is: {UserPreferences.BackColor}");
  }
}
```

编译并运行这段代码，会得到如下结果：

```
User-preferences: BackColor is: Color Red
```

当然，如果在周末执行上述代码，颜色首选项就是 Green。

## 3.4　结构

前面介绍了类如何封装程序中的对象，也介绍了如何将它们存储在堆中，通过这种方式可以在数据的生存期上获得很大的灵活性，但性能会有一定损失。因为托管堆的优化，这种性能损失较小。但是，有时仅需要一个小的数据结构。此时，类提供的功能多于我们需要的功能，由于性能原因，最好使用结构。看看下面的例子：

```
public class Dimensions
{
  public Dimensions(double length, double width)
  {
    Length = length;
    Width = width;
  }
  public double Length { get; }
  public double Width { get; }
}
```

上面的代码定义了类 Dimensions，它只存储了某一项的长度和宽度。假定编写一个布置家具的程序，让人们试着在计算机上重新布置家具，并存储每件家具的尺寸。表面看来使字段变为公共字段会违背编程规则，但这里的关键是我们实际上并不需要类的全部功能。现在只有两个数字，把它们当成一对来处理，要比单个处理方便一些。既不需要很多方法，也不需要从类中继承，也不希望.NET 运行库在堆中遇到麻烦和性能问题，只需要存储两个 double 类型的数据即可。

为此，只需要修改代码，用关键字 struct 代替 class，定义一个结构而不是类，如本章前面所述：

```
public struct Dimensions
{
  public Dimensions(double length, double width)
  {
    Length = length;
    Width = width;
  }

  public double Length { get; }
  public double Width { get; }
}
```

为结构定义函数与为类定义函数完全相同。前面介绍了 Dimensions 结构的构造函数。下面的代码添加 Diagonal 属性，以调用 Math 类的 Sqrt 方法(代码文件 StructsSample/Dimension.cs)：

```
public struct Dimensions
{
  public double Length { get; }
  public double Width { get; }
  public Dimensions(double length, double width)
  {
    Length = length;
    Width = width;
  }
```

```
    public double Diagonal => Math.Sqrt(Length * Length + Width * Width);
}
```

结构是值类型，不是引用类型。它们存储在栈中或存储为内联(如果它们是存储在堆中的另一个对象的一部分)，其生存期的限制与简单的数据类型一样。

- 结构不支持继承。
- 对于结构，构造函数的工作方式有一些区别。如果没有提供默认的构造函数，编译器会自动提供一个，把成员初始化为其默认值。
- 使用结构，可以指定字段如何在内存中布局(第 16 章在介绍特性时将详细论述这个问题)。

因为结构实际上是把数据项组合在一起，所以有时大多数或者全部字段都声明为public。严格来说，这与编写.NET代码的规则相反——根据Microsoft，字段(除了const字段之外)应总是私有的，并由公有属性封装。但是，对于简单的结构，许多开发人员都认为公有字段是可接受的编程方式。

**注意：**
在后台上，int 类型(System.Int32)是一个具有公共字段的结构。新类型 System.ValueType 是一个包含一个或多个公共字段的结构。ValueTuple 在第 13 章中详细讨论。

下面几节将详细说明类和结构之间的区别。

### 3.4.1 结构是值类型

虽然结构是值类型，但在语法上常常可以把它们当作类来处理。例如，在上面的 Dimensions 类的定义中，可以编写下面的代码：

```
var point = new Dimensions();
point.Length = 3;
point.Width = 6;
```

注意，因为结构是值类型，所以 new 运算符与类和其他引用类型的工作方式不同。new 运算符并不分配堆中的内存，而是只调用相应的构造函数，根据传送给它的参数，初始化所有字段。对于结构，可以编写下述完全合法的代码：

```
Dimensions point;
point.Length = 3;
point.Width = 6;
```

如果 Dimensions 是一个类，就会产生一个编译错误，因为 point 包含一个未初始化的引用——不指向任何地方的一个地址，所以不能给其字段设置值。但对于结构，变量声明实际上是为整个结构在栈中分配空间，所以就可以为它赋值了。但要注意下面的代码会产生一个编译错误，编译器会抱怨用户使用了未初始化的变量：

```
Dimensions point;
double d = point.Length;
```

结构遵循其他数据类型都遵循的规则：在使用前所有元素都必须进行初始化。在结构上调用 new 运算符，或者给所有的字段分别赋值，结构就完全初始化了。当然，如果结构定义为类的成员字段，在初始化包含的对象时，该结构会自动初始化为 0。

结构会影响性能的值类型，但根据使用结构的方式，这种影响可能是正面的，也可能是负面的。正面的影响是为结构分配内存时，速度非常快，因为它们将内联或者保存在栈中。在结构超出了作用域被删除时，速度也很快，不需要等待垃圾收集。负面影响是，只要把结构作为参数来传递或者把一个结构赋予另一个结构(如 A=B，其中 A 和 B 是结构)，结构的所有内容就被复制，而对于类，则只复制引用。这样就会有性能损失，根据结构的大小，性能损失也不同。注意，结构主要用于小的数据结构。

但当把结构作为参数传递给方法时，应把它作为 ref 参数传递，以避免性能损失——此时只传递了结构在内存中的地址，这样传递速度就与在类中的传递速度一样快了。但如果这样做，就必须注意被调用的方法可以改变结构的值。详见本章后面的内容。

### 3.4.2　只读结构

从属性中返回一个值类型时，调用方会收到一个副本。设置此值类型的属性只更改副本，原始值不变。这可能会让访问属性的开发人员感到困惑。这就是为什么结构的指导原则定义了值类型应该是不可变的。当然，这个准则对于所有值类型都无效，因为 int、short、double……不是不可变的，而且 ValueTuple 也不是不可变的。然而，大多数结构类型都是不可变的。

使用 C# 7.2 时，readonly 修饰符可以应用于结构，因此编译器保证结构体的不变性。使用 C# 7.2 时，可以声明前面定义的类型 Dimensions 为 readonly，因为它只包含一个修改其成员的构造函数。属性只包含一个 get 访问器，因此不可能进行更改(代码文件 ReadOnlyStructSample/Dimensions.cs)：

```
public readonly struct Dimensions
{
  public double Length { get; }
  public double Width { get; }

  public Dimensions(double length, double width)
  {
    Length = length;
    Width = width;
  }

  public double Diagonal => Math.Sqrt(Length * Length + Width * Width);
}
```

对于 readonly 修饰符，如果在创建对象后类型更改了字段或属性，编译器就会报错。使用这个修饰符，编译器可以生成优化的代码，使其在传递结构体时不会复制结构的内容；相反，编译器使用引用，因为它永远不会改变。

### 3.4.3　结构和继承

结构不是为继承设计的。这意味着：它不能从一个结构中继承。唯一的例外是对应的结构(和 C#中的其他类型一样)最终派生于类 System.Object。因此，结构也可以访问 System.Object 的方法。在结构中，甚至可以重写 System.Object 中的方法——如重写 ToString()方法。结构的继承链是：每个结构派生自 System.ValueType 类，System.ValueType 类又派生自 System.Object。ValueType 并没有给 Object 添加任何新成员，但提供了一些更适合结构的实现方式。注意，不能为结构提供其他基类：每个结构都派生自 ValueType。

**注意：**

只有结构作为对象时，才从 System.ValueType 中继承。不能用作对象的结构是引用结构。这些类型自 C# 7.2 以来一直可用。这个特性参见稍后的“ref 结构”。

**注意：**

要比较结构值，最好实现接口 IEquatable<T>。该接口将在第 6 章中讨论。

### 3.4.4　结构的构造函数

为结构定义构造函数的方式与为类定义构造函数的方式相同。

前面说过，默认构造函数把数值字段都初始化为 0，且总是隐式地给出，即使提供了其他带参数的构造函数，也是如此。不能为结构创建定制的默认构造函数。

```
public Dimensions(double length, double width)
{
  Length = length;
  Width = width;
}
```

另外，可以像类那样为结构提供 Close()或 Dispose()方法。第 17 章将讨论 Dispose()方法。

### 3.4.5 ref 结构

结构并不总是放在堆栈上。它们也可以放在堆上。可以为对象分配一个结构体，这会在堆中创建一个对象。这种行为在某些类型中可能是一个问题。在.NET Core 2.1 中，Span 类型允许访问堆栈上的内存。Span 类型的副本需要是原子的。只有当类型放在堆栈上时，才能保证这一点。此外，Span 类型可以在其字段中使用托管指针。在堆上有这样的指针，可以在垃圾收集器运行时使应用程序崩溃。因此，需要保证类型放在堆栈上。

使用新的 C# 7.2 语言结构，引用类型存储在堆上，值类型通常存储在栈上，但也可以存储在堆上。还有第三种可用类型，即只能在栈上存储的值类型。

此类型是将 ref 修饰符应用于结构而创建的，如下面的代码片段所示。可以添加属性、值字段、引用类型和方法——就像其他结构一样(代码文件 RefStructSample/ValueTypeOnly.cs)：

```
ref struct ValueTypeOnly
{
  //...
}
```

这种类型不能执行的操作是将它分配给对象——例如，调用 Object 基类的方法(如 ToString)。这将导致装箱，并创建一个引用类型，这种类型是不允许这种操作的。

**注意：**

对于大多数应用程序，不需要创建自定义 ref struct 类型。但是，对于需要减少垃圾收集的高性能应用程序，需要使用这种类型。要获得 ref struct 的更多信息，使用这种类型的原因，以及使用 ref return 和 ref local，应该阅读第 17 章，其中详细介绍了 Span 类型，以及关于 ref 的更多信息。

## 3.5 按值和按引用传递参数

假设有一个类型 A，它有一个 int 类型的属性 X。ChangeA 方法接收类型 A 的参数，把 X 的值改为 2(代码文件 PassingByValueAndReference/Program.cs)：

```
public static void ChangeA(A a)
{
  a.X = 2;
}
```

Main()方法创建类型 A 的实例，把 X 初始化为 1，调用 ChangeA 方法：

```
static void Main()
{
  A a1 = new A { X = 1 };
  ChangeA(a1);
  Console.WriteLine($"a1.X: {a1.X}");
}
```

输出是什么？1 还是 2？

答案视情况而定。需要知道 A 是一个类还是结构。下面先假定 A 是结构：

```
public struct A
{
  public int X { get; set; }
}
```

结构按值传递，通过按值传递，ChangeA 方法中的变量 a 得到堆栈中变量 a1 的一个副本。在方法 ChangeA 的最后修改并销毁副本。a1 的内容从不改变，一直是 1。

A 作为一个类时，是完全不同的：

```
public class A
{
  public int X { get; set; }
}
```

类按引用传递。这样，a 变量把堆上的同一个对象引用为变量 a1。当 ChangeA 修改 a 的 X 属性值时，把它改为 a1.X，因为它是同一个对象。这里，结果是 2。

**注意：**

为了避免在更改成员时类和结构之间的不同行为上出现这种混淆，最好将结构设置为不可变的。如果一个结构体只有不允许改变状态的成员，就不会陷入如此混乱的境地。当然，使 struct 类型不可变的规则总是有例外的。C# 7 中新增的 ValueTuple 实现为一个可变结构体。然而，使用 ValueTuple，公共成员就是字段，而不是属性(这是提供公共字段的准则的另一个例外)。由于元组很重要，且以 int 和 float 的方式使用它们，这是违反一些指导原则的好理由。

### 3.5.1 ref 参数

也可以通过引用传递结构。如果 A 是结构类型，就添加 ref 修饰符，修改 ChangeA 方法的声明，通过引用传递变量：

```
public static void ChangeA(ref A a)
{
  a.X = 2;
}
```

从调用端也可以看出这一点，所以给方法参数应用了 ref 修饰符后，在调用方法时需要添加它：

```
static void Main()
{
  A a1 = new A { X = 1 };
  ChangeA(ref a1);
  Console.WriteLine($"a1.X: {a1.X}");
}
```

现在，与类类型一样，结构也按引用传递，所以结果是 2。

类类型如何使用 ref 修饰符？下面修改 ChangeA 方法的实现：

```
public static void ChangeA(A a)
{
  a.X = 2;
  a = new A { X = 3 };
}
```

使用 A 类型的类，可以预期什么结果？当然，Main()方法的结果不是 1，因为按引用传递是通过类类型实现的。a.X 设置为 2，就改变了原始对象 a1。然而，下一行 a = new A { X = 3 }现在在堆上创建一个新对象，和一个对新对象的引用。Main()方法中使用的变量 a1 仍然引用值为 2 的旧对象。ChangeA 方法结束后，没有引用堆上的新对象，可以回收它。所以这里的结果是 2。

把 A 作为类类型，使用 ref 修饰符，传递对引用的引用(在 C++术语中，是一个指向指针的指针)，它允许分配一个新对象，Main()方法显示了结果 3：

```
public static void ChangeA(ref A a)
{
  a.X = 2;
  a = new A { X = 3 };
}
```

最后，一定要理解，C#对传递给方法的参数继续应用初始化要求。在任何变量传递给方法之前，必须初始化，无论是按值还是按引用传递。

**注意：**

在 C# 7 中，还可以对局部变量和方法的返回类型使用 ref 关键字。这个新特性在第 17 章中讨论。

### 3.5.2 out 参数

如果方法返回一个值，该方法通常声明返回类型，并返回结果。如果方法返回多个值，可能类型还不同，该怎么办？这有不同的选项。一个选项是声明类和结构，把应该返回的所有信息都定义为该类型的成员。另一个选项是使用元组类型。元组参见第 13 章。第三个选项是使用 out 关键字。

下面的例子使用通过 Int32 类型定义的 Parse 方法。ReadLine 方法获取用户输入的字符串。假设用户输入一

个数字，int.Parse 方法把它转换为字符串，并返回该数字(代码文件 OutKeywordSample/Program.cs)：

```
string input1 = Console.ReadLine();
int result1 = int.Parse(input1);
Console.WriteLine($"result: {result1}");
```

然而，用户并不总是输入希望他们输入的数据。如果用户没有输入数字，就会抛出一个异常。当然，可以捕获异常，并相应地处理用户，但“正常”情况不这么做。也许可以认为，“正常”情况就是用户输入了错误的数据。处理异常参见第 14 章。

要处理类型错误的数据，更好的方法是使用 Int32 类型的另一个方法：TryParse。TryParse 声明为无论解析成功与否，都返回一个 bool 类型。解析的结果(如果成功)是使用 out 修饰符返回一个参数：

```
public static bool TryParse(string s, out int result);
```

调用此方法后，result 变量不需要预先初始化，而是在方法中初始化变量。在 C# 7 中，也可以在方法调用时声明变量。与 ref 关键字类似，在调用方法时需要提供 out 关键字，而不仅仅在声明方法时提供：

```
string input2 = ReadLine();
if (int.TryParse(input2, out int result2))
{
  Console.WriteLine($"result: {result2}");
}
else
{
  Console.WriteLine("not a number");
}
```

**注意：**

out var 是 C# 7 的一个新特性。在 C# 7 之前，需要在调用该方法之前声明一个 out 变量。在 C# 7 中，可以调用方法来实现声明。如果类型是由方法签名明确定义的，则可以使用 var 关键字(这就是为什么 out var 知道这个特性的原因)来声明变量。还可以定义具体的类型，如前面的代码片段所示。该变量的作用域在方法调用之后是有效的。

### 3.5.3 in 参数

C# 7.2 向参数添加了 in 修饰符。out 修饰符允许返回参数指定的值。in 修饰符保证发送到方法中的数据不会更改(在传递值类型时)。

下面定义一个简单的可变结构体，名称为 AValueType，再定义一个公共可变字段(代码文件 InParameterSample/AValueType.cs)：

```
struct AValueType
{
  public int Data;
}
```

现在，使用 in 修饰符定义一个方法时，变量就不能更改了。试图更改可变字段 Data，编译器会抱怨不能为只读变量的成员分配值，因为该变量是只读的。in 修饰符使参数设置为只读变量(代码文件 InParameterSample/Program.cs)：

```
static void CantChange(in AValueType a)
{
  // a.Data = 43;  // does not compile - readonly variable
  Console.WriteLine(a.Data);
}
```

当调用方法 CantChange 时，可以通过传递或不传递 in 修饰符来调用该方法。这对生成的代码没有影响。

使用值类型和 in 修饰符，不仅有助于确保不更改内存，编译器还可以创建更好的优化代码。与使用方法调用来复制值类型不同，编译器可以使用引用，从而减少所需的内存并提高性能。

**注意：**

in 修饰符主要用于值类型。也可以对引用类型使用它。in 修饰符用于引用类型时，可以更改变量的内容，但不能更改变量本身。

# 3.6　可空类型

引用类型(类)的变量可以为空，而值类型(结构)的变量不能。在一些情况下，这可能是一个问题，如把 C#类型映射到数据库或 XML 类型。数据库或 XML 数量可以为空，而 int 或 double 不能为空。

处理这个冲突的一个方法是使用映射到数据库数字类型的类(这由 Java 实现)。使用引用类型，映射到允许空值的数据库数字，有一个重要的缺点：它带来了额外的开销。对于引用类型，需要垃圾收集器进行清理。值类型不需要用垃圾收集器清理；变量超出作用域时，从内存中删除。

C#有一个解决方案：可空类型。可空类型是可以为空的值类型。可空类型只需要在类型的后面添加“?”(它必须是结构)。与基本结构相比，值类型唯一的开销是一个可以确定它是否为空的布尔成员。

在下面的代码片段中，x1 是一个普通的 int，x2 是一个可以为空的 int。因为 x2 是可以为空的 int，所以可以把 null 分配给 x2：

```
int x1 = 1;
int? x2 = null;
```

因为 int 值可以分配给 int?，所以给 int?传递一个 int 变量总是会成功，编译器会接受它：

```
int? x3 = x1;
```

反过来是不正确的。int?不能直接分配给 int。这可能失败，因此需要一个类型转换：

```
int x4 = (int)x3;
```

当然，如果 x3 是 null，类型转换操作就会生成一个异常。更好的方法是使用可空类型的 HasValue 和 Value 属性。HasValue 返回 true 或 false，这取决于可空类型是否有值，Value 返回底层的值。使用条件操作符填充 x5，不会抛出异常。如果 x3 是 null，HasValue 就返回 false，这里给变量 x5 提供-1：

```
int x5 = x3.HasValue ? x3.Value : -1;
```

使用合并操作符??，可空类型可以使用较短的语法。如果 x3 是 null，则用变量 x6 给它设置-1，否则提取 x3 的值：

```
int x6 = x3 ?? -1;
```

**注意：**

对于可空类型，可以使用能用于基本类型的所有可用操作符，例如，可用于 int?的+、-、*、/等。每个结构类型都可以使用可空类型，而不仅是预定义的 C#类型。可空类型及其后台的内容参见第 5 章。

# 3.7　枚举类型

枚举是一个值类型，包含一组命名的常量，如这里的 Color 类型。枚举类型用 enum 关键字定义(代码文件 EnumSample/Color. cs)：

```
public enum Color
{
  Red,
  Green,
  Blue
}
```

可以声明枚举类型的变量，如变量 c1，用枚举类型的名称作为前缀设置一个命名常量，来赋予枚举中的一个值(代码文件 EnumSample/Program.cs)：

```
private static void ColorSamples()
{
  Color c1 = Color.Red;
  Console.WriteLine(c1);
  //...
}
```

运行程序，控制台输出显示 Red，这是枚举的常量值。

默认情况下，enum 的类型是 int。这个基本类型可以改为其他整数类型(byte、short、int、带符号的 long 和无符号变量)。命名常量的值从 0 开始递增，但它们可以改为其他值：

```
public enum Color : short
{
  Red = 1,
  Green = 2,
  Blue = 3
}
```

使用强制类型转换可以把数字改为枚举值，把枚举值改为数字。

```
Color c2 = (Color)2;
short number = (short)c2;
```

还可以使用 enum 类型把多个选项分配给一个变量，而不仅仅是一个枚举常量。为此，分配给常量的值必须是不同的位，Flags 属性需要用枚举设置。

枚举类型 DaysOfWeek 为每天定义了不同的值。要设置不同的位，可以使用用 0x 前缀指定的十六进制值轻松地完成，Flags 属性是编译器创建值的另一个字符串表示的信息，例如给 DaysOfWeek 的一个变量设置值 3，结果是 Monday，如果使用 Flags 属性，结果就是 Tuesday(代码文件 EnumSample/DaysOfWeek.cs)：

```
[Flags]
public enum DaysOfWeek
{
  Monday = 0x1,
  Tuesday = 0x2,
  Wednesday = 0x4,
  Thursday = 0x8,
  Friday = 0x10,
  Saturday = 0x20,
  Sunday = 0x40
}
```

有了这个枚举声明，就可以使用“逻辑或”运算符为一个变量指定多个值（代码文件 EnumSample/Program.cs)：

```
DaysOfWeek mondayAndWednesday = DaysOfWeek.Monday | DaysOfWeek.Wednesday;
Console.WriteLine(mondayAndWednesday);
```

运行程序，输出是日期的字符串表示：

```
Monday, Tuesday
```

设置不同的位，也可以结合单个位来包括多个值，如 Weekend 的值 0x60 是用“逻辑或”运算符结合了 Saturday 和 Sunday。Workday 则结合了从 Monday 到 Friday 的所有日子，AllWeek 用“逻辑或”运算符结合了 Workday 和 Weekend (代码文件 EnumSample/DaysOfWeek.cs)：

```
[Flags]
public enum DaysOfWeek
{
  Monday = 0x1,
  Tuesday = 0x2,
  Wednesday = 0x4,
  Thursday = 0x8,
  Friday = 0x10,
  Saturday = 0x20,
  Sunday = 0x40,
  Weekend = Saturday | Sunday
  Workday = 0x1f,
  AllWeek = Workday | Weekend
}
```

有了这些代码，就可以把 DaysOfWeek.Weekend 直接分配给变量，指定用“逻辑或”运算符结合 DaysOfWeek.Saturday 和 DaysOfWeek.Sunday 的单个值，也可以得到相同的结果。输出会显示 Weekend 的字符串表示。

```
DaysOfWeek weekend = DaysOfWeek.Saturday | DaysOfWeek.Sunday;
Console.WriteLine(weekend);
```

使用枚举，类 Enum 有时非常有助于动态获得枚举类型的信息。枚举提供了方法来解析字符串，获得相应

的枚举常数，获得枚举类型的所有名称和值。

下面的代码片段使用字符串和 Enum.TryParse 来获得相应的 Color 值(代码文件 EnumSample/Program.cs)：

```
Color red;
if (Enum.TryParse<Color>("Red", out red))
{
  Console.WriteLine($"successfully parsed {red}");
}
```

> **注意：**
> Enum.TryParse < T >()是一个泛型方法，其中 T 是泛型参数类型。这个参数类型需要用方法调用定义。泛型方法参见第 5 章。

Enum.GetNames 方法返回一个包含所有枚举名的字符串数组：

```
foreach (var day in Enum.GetNames(typeof(Color)))
{
  Console.WriteLine(day);
}
```

运行应用程序，输出如下：

```
Red
Green
Blue
```

为了获得枚举的所有值，可以使用方法 Enum.GetValues。Enum.GetValues 返回枚举值的一个数组。为了获得整数值，需要把它转换为枚举的底层类型，为此应使用 foreach 语句：

```
foreach (short val in Enum.GetValues(typeof(Color)))
{
  Console.WriteLine(val);
}
```

## 3.8　部分类

partial 关键字允许把类、结构、方法或接口放在多个文件中。一般情况下，某种类型的代码生成器生成了一个类的某部分，所以把类放在多个文件中是有益的。假定要给类添加一些从工具中自动生成的内容。如果重新运行该工具，前面所做的修改就会丢失。partial 关键字有助于把类分开放在两个文件中，而对不由代码生成器定义的文件进行修改。

partial 关键字的用法是：把 partial 放在 class、struct 或 interface 关键字的前面。在下面的例子中，SampleClass 类驻留在两个不同的源文件 SampleClassAutogenerated.cs 和 SampleClass.cs 中：

```
SampleClass.cs:
//SampleClassAutogenerated.cs
partial class SampleClass
{
  public void MethodOne() { }
}

//SampleClass.cs
partial class SampleClass
{
  public void MethodTwo() { }
}
```

当编译包含这两个源文件的项目时，会创建一个 SampleClass 类，它有两个方法 MethodOne()和 MethodTwo()。

如果声明类时使用了下面的关键字，则这些关键字就必须应用于同一个类的所有部分：

- public
- private
- protected

- internal
- abstract
- sealed
- new
- 一般约束

在嵌套的类型中，只要 partial 关键字位于 class 关键字的前面，就可以嵌套部分类。在把部分类编译到类型中时，属性、XML 注释、接口、泛型类型的参数属性和成员会合并。有如下两个源文件：

```
// SampleClassAutogenerated.cs
[CustomAttribute]
partial class SampleClass: SampleBaseClass, ISampleClass
{
  public void MethodOne() { }
}

// SampleClass.cs
[AnotherAttribute]
partial class SampleClass: IOtherSampleClass
{
  public void MethodTwo() { }
}
```

编译后，等价的源文件变成：

```
[CustomAttribute]
[AnotherAttribute]
partial class SampleClass: SampleBaseClass, ISampleClass, IOtherSampleClass
{
  public void MethodOne() { }
  public void MethodTwo() { }
}
```

**注意：**

尽管 partial 关键字很容易创建跨多个文件的庞大的类，且不同的开发人员处理同一个类的不同文件，但该关键字并不用于这个目的。在这种情况下，最好把大类拆分成几个小类，一个类只用于一个目的。

部分类可以包含部分方法。如果生成的代码应该调用可能不存在的方法，这就是非常有用的。扩展部分类的程序员可以决定创建部分方法的自定义实现代码，或者什么也不做。下面的代码片段包含一个部分类，其方法 MethodOne 调用 APartialMethod 方法。APartialMethod 方法用 partial 关键字声明，因此不需要任何实现代码。如果没有实现代码，编译器将删除这个方法调用：

```
//SampleClassAutogenerated.cs
partial class SampleClass
{
  public void MethodOne()
  {
    APartialMethod();
  }
  public partial void APartialMethod();
}
```

部分方法的实现可以放在部分类的任何其他地方，如下面的代码片段所示。有了这个方法，编译器就在 MethodOne 内创建代码，调用这里声明的 APartialMethod：

```
// SampleClass.cs
partial class SampleClass: IOtherSampleClass
{
  public void APartialMethod()
  {
    // implementation of APartialMethod
  }
}
```

部分方法必须是 void 类型，否则编译器在没有实现代码的情况下无法删除调用。

# 3.9 扩展方法

有许多扩展类的方式。继承(参见第 4 章)就是给对象添加功能的好方法。扩展方法是给对象添加功能的另一个选项，在不能使用继承时，也可以使用这个选项(例如类是密封的)。

**注意：**
扩展方法也可以用于扩展接口。这样，实现该接口的所有类就有了公共功能。接口参见第 4 章。

扩展方法是静态方法，它是类的一部分，但实际上没有放在类的源代码中。

假设希望用一个方法扩展 string 类型，该方法计算字符串中的单词数。GetWordCount 方法利用 String.Split 方法把字符串分割到字符串数组中，使用 Length 属性计算数组中元素的个数(代码文件 ExtensionMethods/Program.cs)：

```
ExtensionMethods/Program.cs):
public static class StringExtension
{
  public static int GetWordCount(this string s) => s.Split().Length;
}
```

使用 this 关键字和第一个参数来扩展字符串。这个关键字定义了要扩展的类型。

即使扩展方法是静态的，也要使用标准的实例方法语法。注意，这里使用 fox 变量而没有使用类型名来调用 GetWordCount ()。

```
string fox = "the quick brown fox jumped over the lazy dogs down " +
"9876543210 times";
int wordCount = fox.GetWordCount();
Console.WriteLine($"{wordCount} words");
```

在后台，编译器把它改为调用静态方法：

```
int wordCount = StringExtension.GetWordCount(fox);
```

使用实例方法的语法，而不是从代码中直接调用静态方法，会得到一个好得多的语法。这个语法还有一个好处：该方法的实现可以用另一个类取代，而不需要更改代码——只需要运行新的编译器。

编译器如何找到某个类型的扩展方法？this 关键字必须匹配类型的扩展方法，而且需要打开定义扩展方法的静态类所在的名称空间。如果把 StringExtensions 类放在名称空间 Wrox.Extensions 中，则只有用 using 指令打开 Wrox.Extensions，编译器才能找到 GetWordCount 方法。如果类型还定义了同名的实例方法，扩展方法就永远不会使用。类中已有的任何实例方法都优先。当多个同名的扩展方法扩展相同的类型，打开所有这些类型的名称空间时，编译器会产生一个错误，指出调用是模棱两可的，它不能决定在多个实现代码中选择哪个。然而，如果调用代码在一个名称空间中，这个名称空间就优先。

**注意：**
语言集成查询(Language Integrated Query，LINQ)利用了许多扩展方法。LINQ 参见第 12 章。

# 3.10 Object 类

前面提到，所有的.NET 类最终都派生自 System.Object。实际上，如果在定义类时没有指定基类，编译器就会自动假定这个类派生自 Object。本章没有使用继承，所以前面介绍的每个类都派生自 System.Object(如前所述，对于结构，这个派生是间接的：结构总是派生自 System.ValueType，System.ValueType 又派生自 System.Object)。

其实际意义在于，除了自己定义的方法和属性等外，还可以访问为 Object 类定义的许多公有的和受保护的成员方法。这些方法可用于自己定义的所有其他类中。

下面将简要总结每个方法的作用：

- **ToString()方法**：是获取对象的字符串表示的一种便捷方式。当只需要快速获取对象的内容，以进行调试时，就可以使用这个方法。在数据的格式化方面，它几乎没有提供选择：例如，在原则上日期可以表示为许多不同的格式，但 DateTime.ToString()没有在这方面提供任何选择。如果需要更复杂的字符串表示，例如，考虑用户的格式化首选项或区域性，就应实现 IFormattable 接口(参见第 9 章)。
- **GetHashCode()方法**：如果对象放在名为映射(也称为散列表或字典)的数据结构中，就可以使用这个方法。处理这些结构的类使用该方法确定把对象放在结构的什么地方。如果希望把类用作字典的一个键，就需要重写 GetHashCode()方法。实现该方法重载的方式有一些相当严格的限制，这些将在第 10 章介绍字典时讨论。
- **Equals()(两个版本)和 ReferenceEquals()方法**：注意有 3 个用于比较对象相等性的不同方法，这说明.NET Framework 在比较相等性方面有相当复杂的模式。这 3 个方法和比较运算符“==”在使用方式上有微妙的区别。而且，在重写带一个参数的虚 Equals()方法时也有一些限制，因为 System.Collections 名称空间中的一些基类要调用该方法，并希望它以特定的方式执行。第 6 章在介绍运算符时将探讨这些方法的使用。
- **Finalize()方法**：第 17 章将介绍这个方法，它最接近 C++风格的析构函数，在引用对象作为垃圾被收集以清理资源时调用它。Object 中实现的 Finalize()方法实际上什么也没有做，因而被垃圾收集器忽略。如果对象拥有对非托管资源的引用，则在该对象被删除时，就需要删除这些引用，此时一般要重写 Finalize()。垃圾收集器不能直接删除这些对非托管资源的引用，因为它只负责托管的资源，于是它只能依赖用户提供的 Finalize()。
- **GetType()方法**：这个方法返回从 System.Type 派生的类的一个实例，因此可以提供对象成员所属类的更多信息，包括基本类型、方法、属性等。System.Type 还提供了.NET 的反射技术的入口点。这个主题详见第 16 章。
- **MemberwiseClone()方法**：这是 System.Object 中唯一没有在本书的其他地方详细论述的方法。不需要讨论这个方法，因为它在概念上相当简单，它只复制对象，并返回对副本的一个引用(对于值类型，就是一个装箱的引用)。注意，得到的副本是一个浅表复制，即它复制了类中的所有值类型。如果类包含内嵌的引用，就只复制引用，而不复制引用的对象。这个方法是受保护的，所以不能用于复制外部的对象。该方法不是虚方法，所以不能重写它的实现代码。

## 3.11 小结

本章介绍了 C#中声明和处理对象的语法，论述了如何声明静态和实例字段、属性、方法和构造函数。还讨论了 C# 7 中新增的特性。例如，表达式体的成员和构造函数、属性访问器和 out 变量。

我们还阐述了 C#中的所有类型最终都派生自类 System.Object，这说明所有的类型都开始于一组基本的实用方法，包括 ToString()。

本章多次提到了继承，第 4 章将介绍 C#中的实现(implementation)继承、接口继承和面向对象的其他方面。

# 第 4 章

# 继　　承

**本章要点**

- 继承的类型
- 实现继承
- 访问修饰符
- 接口
- is 和 as 运算符

**本章源代码下载地址(wrox.com)：**

打开 www.wrox.com 的 Download Code 选项卡可下载本章源代码。源代码也可以在 ObjectOrientation 目录的 https://github.com/ProfessionalCSharp/ProfessionalCSharp7 中找到。本章代码分为以下几个主要的示例文件：

- VirtualMethods
- InheritanceWithConstructors
- UsingInterfaces

## 4.1 面向对象

C#不是一种纯粹的面向对象编程语言。C#提供了多种编程范例。然而，面向对象是 C#的一个重要概念，也是.NET 提供的所有库的核心原则。

面向对象的三个最重要的概念是继承、封装和多态性。第 3 章谈到如何创建单独的类，来安排属性、方法和字段。当某类型的成员声明为 private 时，它们就不能从外部访问。它们封装在类型中。本章的重点是继承和多态性。

第 3 章提到，所有类最终都派生于 System.Object。本章介绍如何创建类的层次结构，多态性如何应用于 C#，还描述与继承相关的所有 C#关键字。

## 4.2 继承的类型

首先介绍一些面向对象(Object-Oriented，OO)术语，看看 C#在继承方面支持和不支持的功能。

- **单重继承：**表示一个类可以派生自一个基类。C#就采用这种继承。
- **多重继承：**多重继承允许一个类派生自多个类。C#不支持类的多重继承，但允许接口的多重继承。
- **多层继承：**多层继承允许继承有更大的层次结构。类 B 派生自类 A，类 C 又派生自类 B。其中，类 B 也称为中间基类，C#支持它，也很常用。
- **接口继承：**定义了接口的继承。这里允许多重继承。接口和接口继承参见本章后面的“接口”一节。

下面讨论继承和 C#的某些特定问题。

### 4.2.1 多重继承

一些语言(如 C++)支持所谓的“多重继承”，即一个类派生自多个类。对于实现继承，多重继承会给生成的代码增加复杂性，还会带来一些开销。因此，C#的设计人员决定不支持类的多重继承，因为支持多重继承会增加复杂性，还会带来一些开销。

而 C#又允许类型派生自多个接口。一个类型可以实现多个接口。这说明，C#类可以派生自另一个类和任意多个接口。更准确地说，因为 System.Object 是一个公共的基类，所以每个 C#类(除了 Object 类之外)都有一个基类，还可以有任意多个基接口。

### 4.2.2 结构和类

第 3 章区分了结构(值类型)和类(引用类型)。使用结构的一个限制是结构不支持继承，但每个结构都自动派生自 System.ValueType。不能编码实现结构的类型层次，但结构可以实现接口。换言之，结构并不支持实现继承，但支持接口继承。定义的结构和类可以总结为：

- 结构总是派生自 System.ValueType，它们还可以派生自任意多个接口。
- 类总是派生自 System.Object 或用户选择的另一个类，它们还可以派生自任意多个接口。

## 4.3 实现继承

如果要声明派生自另一个类的一个类，就可以使用下面的语法：

```
class MyDerivedClass: MyBaseClass
{
  // members
}
```

如果类(或结构)也派生自接口，则用逗号分隔列表中的基类和接口：

```
public class MyDerivedClass: MyBaseClass, IInterface1, IInterface2
{
  // members
}
```

**注意：**
如果类和接口都用于派生，则类总是必须放在接口的前面。

对于结构，语法如下(只能用于接口继承)：

```
public struct MyDerivedStruct: IInterface1, IInterface2
{
  // members
}
```

如果在类定义中没有指定基类，C#编译器就假定 System.Object 是基类。因此，派生自 Object 类(或使用 object 关键字)，与不定义基类的效果是相同的。

```
class MyClass // implicitly derives from System.Object
{
  // members
}
```

下面的例子定义了基类 Shape。无论是矩形还是椭圆，形状都有一些共同点：形状都有位置和大小。定义相应的类时，位置和大小应包含在 Shape 类中。Shape 类定义了只读属性 Position 和 Shape，它们使用自动属性初始化器来初始化(代码文件 VirtualMethods /Shape.cs)：

```
public class Position
{
  public int X { get; set; }
  public int Y { get; set; }
}

public class Size
{
  public int Width { get; set; }
  public int Height { get; set; }
}

public class Shape
{
  public Position Position { get; } = new Position();
  public Size Size { get; } = new Size();
}
```

### 4.3.1　虚方法

把一个基类方法声明为 virtual，就可以在任何派生类中重写该方法：

```
public class Shape
{
  public virtual void Draw() =>
    Console.WriteLine($"Shape with {Position} and {Size}");
}
```

如果实现代码只有一行，也可以把 virtual 关键字和表达式体的方法(使用 lambda 运算符)一起使用。这个语法可以独立于修饰符，单独使用：

```
public class Shape
{
  public virtual void Draw() =>
    Console.WriteLine($"Shape with {Position} and {Size}");
}
```

也可以把属性声明为 virtual。对于虚属性或重写属性，语法与非虚属性相同，但要在定义中添加关键字 virtual，其语法如下所示：

```
public virtual Size Size { get; set; }
```

当然，也可以给虚属性使用完整的属性语法。下面的代码片段使用了 C# 7 表达式体的属性访问器：

```
private Size _size;
public virtual Size Size
{
  get => _size;
  set => _size = value;
}
```

为简单起见，下面的讨论将主要集中于方法，但其规则也适用于属性。

C#中虚函数的概念与标准 OOP 的概念相同：可以在派生类中重写虚函数。在调用方法时，会调用该类对象的合适方法。在 C#中，函数在默认情况下不是虚拟的，但(除了构造函数以外)可以显式地声明为 virtual。这遵循 C++的方式，即从性能的角度来看，除非显式指定，否则函数就不是虚拟的。而在 Java 中，所有的函数都是虚拟的。但 C#的语法与 C++的语法不同，因为 C#要求在派生类的函数重写另一个函数时，要使用 override 关键字显式声明(代码文件 VirtualMethods/ConcreteShapes.cs)：

```
public class Rectangle : Shape
{
  public override void Draw() =>
    Console.WriteLine($"Rectangle with {Position} and {Size}");
}
```

重写方法的语法避免了 C++中很容易发生的潜在运行错误：当派生类的方法签名无意中与基类版本略有差

别时，该方法就不能重写基类的方法。在 C#中，这会出现一个编译错误，因为编译器会认为函数已标记为 override，但没有重写其基类的方法。

Size 和 Position 类型重写了 ToString()方法。这个方法在基类 Object 中声明为 virtual：

```
public class Position
{
  public int X { get; set; }
  public int Y { get; set; }
  public override string ToString() => $"X: {X}, Y: {Y}";
}

public class Size
{
  public int Width { get; set; }
  public int Height { get; set; }
  public override string ToString() => $"Width: {Width}, Height: {Height}";
}
```

**注意：**

基类 Object 的成员参见第 3 章。

**注意：**

重写基类的方法时，签名(所有参数类型和方法名)和返回类型必须完全匹配。否则，以后创建的新成员就不覆盖基类成员。

在 Main()方法中，实例化矩形 r，初始化其属性，调用其方法 Draw() (代码文件 VirtualMethods/Program.cs)：

```
var r = new Rectangle();
r.Position.X = 33;
r.Position.Y = 22;
r.Size.Width = 200;
r.Size.Height = 100;
r.Draw();
```

运行程序，查看 Draw()方法的输出：

```
Rectangle with X: 33, Y: 22 and Width: 200, Height: 100
```

成员字段和静态函数都不能声明为 virtual，因为这个概念只对类中的实例函数成员有意义。

## 4.3.2 多态性

使用多态性，可以动态地定义调用的方法，而不是在编译期间定义。编译器创建一个虚拟方法表(vtable)，其中列出了可以在运行期间调用的方法，它根据运行期间的类型调用方法。

在下面的例子中，DrawShape()方法接收一个 Shape 参数，并调用 Shape 类的 Draw()方法(代码文件 VirtualMethods/Program.cs)：

```
public static void DrawShape(Shape shape) => shape.Draw();
```

使用之前创建的矩形调用方法。尽管方法声明为接收一个 Shape 对象，但任何派生 Shape 的类型(包括 Rectangle)都可以传递给这个方法：

```
DrawShape(r);
```

运行这个程序，查看 Rectangle.Draw 方法()而不是 Shape.Draw()方法的输出。输出行从 Rectangle 开始。如果基类的方法不是虚拟方法或没有重写派生类的方法，就使用所声明对象(Shape)的类型的 Draw()方法，因此输出从 Shape 开始：

```
Rectangle with X: 33, Y: 22 and Width: 200, Height: 100
```

### 4.3.3　隐藏方法

如果签名相同的方法在基类和派生类中都进行了声明，但该方法没有分别声明为 virtual 和 override，派生类方法就会隐藏基类方法。

在大多数情况下，是要重写方法，而不是隐藏方法，因为隐藏方法会造成对于给定类的实例调用错误方法的危险。但是，如下面的例子所示，C#语法可以确保开发人员在编译时收到这个潜在错误的警告，从而使隐藏方法(如果这确实是用户的本意)更加安全。这也是类库开发人员得到的版本方面的好处。

假定类库中有一个类 Shape：

```
public class Shape
{
  // various members
}
```

在将来的某一刻，要编写一个派生类 Ellipse，用它给 Shape 基类添加某个功能，特别是要添加该基类中目前没有的方法——MoveBy()：

```
public class Ellipse: Shape
{
  public void MoveBy(int x, int y)
  {
    Position.X += x;
    Position.Y += y;
  }
}
```

过了一段时间，基类的编写者决定扩展基类的功能。为了保持一致，他也添加了一个名为 MoveBy()的方法，该方法的名称和签名与前面添加的方法相同，但并不完成相同的工作。这个新方法可能声明为 virtual，也可能不声明为 virtual。

如果重新编译派生的类，会得到一个编译器警告，因为出现了一个潜在的方法冲突。然而，也可能使用了新的基类，但没有编译派生类；只是替换了基类程序集。基类程序集可以安装在全局程序集缓存中(许多 Framework 程序集都安装在此)。

现在假设基类的 MoveBy()方法声明为虚方法，基类本身调用 MoveBy()方法。会调用哪个方法？基类的方法还是前面定义的派生类的 MoveBy()方法吗？因为派生类的 MoveBy()方法没有用 override 关键字定义(这是不可能的，因为基类 MoveBy()方法以前不存在)，编译器假定派生类的 MoveBy()方法是一个完全不同的方法，与基类的方法没有任何关系，只是名字相同。这种方法的处理方式就好像它有另一个名称一样。

编译 Ellipse 类会生成一个编译警告，提醒使用 new 关键字隐藏方法。在实践中，不使用 new 关键字会得到相同的编译结果，但避免出现编译器警告：

```
public class Ellipse: Shape
{
  new public void Move(Position newPosition)
  {
    Position.X = newPosition.X;
    Position.Y = newPosition.Y;
  }
  //... other members
}
```

不使用 new 关键字，也可以重命名方法，或者，如果基类的方法声明为 virtual，且用作相同的目的，就重写它。然而，如果其他方法已经调用了此方法，简单的重命名会破坏其他代码。

**注意：**

new 方法修饰符不应该故意用于隐藏基类的成员。这个修饰符的主要目的是处理版本冲突，在修改派生类后，响应基类的变化。

### 4.3.4 调用方法的基类版本

C#有一种特殊的语法用于从派生类中调用方法的基类版本：base.<MethodName>()。例如，派生类 Shape 声明了 Move()方法，想要在派生类 Rectangle 中调用它，以使用基类的实现代码。为了添加派生类中的功能，可以使用 base 调用它(代码文件 VirtualMethods/Shape.cs)：

```
public class Shape
{
  public virtual void Move(Position newPosition)
  {
    Position.X = newPosition.X;
    Position.Y = newPosition.Y;
    Console.WriteLine($"moves to {Position}");
  }
  //...other members
}
```

Move()方法在 Rectangle 类中重写，把 Rectangle 一词添加到控制台。写出文本之后，使用 base 关键字调用基类的方法(代码文件 VirtualMethods/ConcreteShapes.cs)：

```
public class Rectangle: Shape
{
  public override void Move(Position newPosition)
  {
    Console.Write("Rectangle ");
    base.Move(newPosition);
  }
  //...other members
}
```

现在，矩形移动到一个新位置(代码文件 VirtualMethods/Program.cs)：

```
r.Move(new Position { X = 120, Y = 40 });
```

运行应用程序，输出是 Rectangle 和 Shape 类中 Move()方法的结果：

```
Rectangle moves to X: 120, Y: 40
```

**注意：**
使用 base 关键字，可以调用基类的任何方法——而不仅仅是已重写的方法。

### 4.3.5 抽象类和抽象方法

C#允许把类和方法声明为 abstract。抽象类不能实例化，而抽象方法不能直接实现，必须在非抽象的派生类中重写。显然，抽象方法本身也是虚拟的(尽管也不需要提供 virtual 关键字，实际上，如果提供了该关键字，就会产生一个语法错误)。如果类包含抽象方法，则该类也是抽象的，也必须声明为抽象的。

下面把 Shape 类改为抽象类。因为其他类需要派生自这个类。新方法 Resize 声明为抽象，因此它不能有在 Shape 类中的任何实现代码(代码文件 VirtualMethods/Shape.cs)：

```
public abstract class Shape
{
  public abstract void Resize(int width, int height); // abstract method
}
```

从抽象基类中派生类型时，需要实现所有抽象成员。否则，编译器会报错：

```
public class Ellipse : Shape
{
  public override void Resize(int width, int height)
  {
    Size.Width = width;
    Size.Height = height;
  }
}
```

当然，实现代码也可以如下面的例子所示。抛出类型 NotImplementationException 的异常也是一种实现方式，在开发过程中，它通常只是一个临时的实现：

```
public override void Resize(int width, int height)
{
  throw new NotImplementedException();
}
```

**注意：**
异常详见第 14 章。

使用抽象的 Shape 类和派生的 Ellipse 类，可以声明 Shape 的一个变量。不能实例化它，但是可以实例化 Ellipse，并将其分配给 Shape 变量(代码文件 VirtualMethods/Program.cs)：

```
Shape s1 = new Ellipse();
DrawShape(s1);
```

### 4.3.6　密封类和密封方法

如果不应创建派生自某个自定义类的类，该自定义类就应密封。给类添加 sealed 修饰符，就不允许创建该类的子类。密封一个方法，表示不能重写该方法。

```
 sealed class FinalClass
{
  //...
}

class DerivedClass: FinalClass // wrong. Cannot derive from sealed class.
{
  //...
}
```

在把类或方法标记为 sealed 时，最可能的情形是：如果在库、类或自己编写的其他类的操作中，类或方法是内部的，则任何尝试重写它的一些功能，都可能导致代码的不稳定。例如，也许没有测试继承，就对继承的设计决策投资。如果是这样，最好把类标记为 sealed。

密封类有另一个原因。对于密封类，编译器知道不能派生类，因此用于虚拟方法的虚拟表可以缩短或消除，以提高性能。string 类是密封的。没有哪个应用程序不使用字符串，最好使这种类型保持最佳性能。把类标记为 sealed 对编译器来说是一个很好的提示。

将一个方法声明为 sealed 的目的类似于一个类。方法可以是基类的重写方法，但是在接下来的例子中，编译器知道，另一个类不能扩展这个方法的虚拟表；它在这里终止继承。

```
class MyClass: MyBaseClass
{
  public sealed override void FinalMethod()
  {
    // implementation
  }
}

class DerivedClass: MyClass
{
  public override void FinalMethod() // wrong. Will give compilation error
  {
  }
}
```

要在方法或属性上使用 sealed 关键字，必须先从基类上把它声明为要重写的方法或属性。如果基类上不希望有重写的方法或属性，就不要把它声明为 virtual。

### 4.3.7　派生类的构造函数

第 3 章介绍了单个类的构造函数是如何工作的。这样，就产生了一个有趣的问题，在开始为层次结构中的类(这个类继承了其他也可能有自定义构造函数的类)定义自己的构造函数时，会发生什么情况？

假定没有为任何类定义任何显式的构造函数，编译器就会为所有的类提供默认的初始化构造函数，在后台会进行许多操作，但编译器可以很好地解决类的层次结构中的所有问题，每个类中的每个字段都会初始化为对

应的默认值。但在添加了一个我们自己的构造函数后，就要通过派生类的层次结构高效地控制构造过程，因此必须确保构造过程顺利进行，不要出现不能按照层次结构进行构造的问题。

为什么派生类会有某些特殊的问题？原因是在创建派生类的实例时，实际上会有多个构造函数起作用。要实例化的类的构造函数本身不能初始化类，还必须调用基类中的构造函数。这就是为什么要通过层次结构进行构造的原因。

在之前的 Shape 类型示例中，使用自动属性初始化器初始化属性：

```
public class Shape
{
  public Position Position { get; } = new Position();
  public Size Size { get; } = new Size();
}
```

在幕后，编译器会给类创建一个默认的构造函数，把属性初始化器放在这个构造函数中：

```
public class Shape
{
  public Shape()
  {
    Position = new Position();
    Size = new Size();
  }

  public Position Position { get; };
  public Size Size { get; };
}
```

当然，实例化派生自 Shape 类的 Rectangle 类型，Rectangle 需要 Position 和 Size，因此在构造派生对象时，调用基类的构造函数。

如果没有在默认构造函数中初始化成员，编译器会自动把引用类型初始化为 null，值类型初始化为 0，布尔类型初始化为 false。布尔类型是值类型，false 与 0 是一样的，所以这个规则也适用于布尔类型。

对于 Ellipse 类，如果基类定义了默认构造函数，只把所有成员初始化为其默认值，就没有必要创建默认的构造函数。当然，仍可以提供一个构造函数，使用构造函数初始化器，调用基构造函数：

```
public class Ellipse : Shape
{
  public Ellipse()
    : base()
  {
  }
}
```

构造函数总是按照层次结构的顺序调用：先调用 System.Object 类的构造函数，再按照层次结构由上向下进行，直到到达编译器要实例化的类为止。为了实例化 Ellipse 类型，先调用 Object 构造函数，再调用 Shape 构造函数，最后调用 Ellipse 构造函数。这些构造函数都处理它自己类中字段的初始化。

现在，改变 Shape 类的构造函数。不是对 Size 和 Position 属性进行默认的初始化，而是在构造函数内赋值(代码文件 InheritanceWithConstructors/Shape.cs )：

```
public abstract class Shape
{
  public Shape(int width, int height, int x, int y)
  {
    Size = new Size { Width = width, Height = height };
    Position = new Position { X = x, Y = y };
  }

  public Position Position { get; }
  public Size Size { get; }
}
```

当删除默认构造函数，重新编译程序时，不能编译 Ellipse 和 Rectangle 类，因为编译器不知道应该把什么值传递给基类唯一的非默认值构造函数。这里需要在派生类中创建一个构造函数，用构造函数初始化器初始化基类构造函数(代码文件 InheritanceWithConstructors/ConcreteShapes.cs )：

```
public Rectangle(int width, int height, int x, int y)
  : base(width, height, x, y)
```

```
{
}
```

把初始化代码放在构造函数块内太迟了，因为基类的构造函数在派生类的构造函数之前调用。这就是为什么在构造函数块之前声明了一个构造函数初始化器。

如果希望允许使用默认的构造函数创建 Rectangle 对象，仍然可以这样做。如果基类的构造函数没有默认的构造函数，也可以这样做，只需要在构造函数初始化器中为基类构造函数指定值，如下所示。在接下来的代码片段中，使用了命名参数，否则很难区分传递的 width、height、x 和 y 值。

```
public Rectangle()
  : base(width: 0, height: 0, x: 0, y: 0)
{
```

**注意：**
命名参数参见第 3 章。

这个过程非常简洁，设计也很合理。每个构造函数都负责处理相应变量的初始化。在这个过程中，正确地实例化了类，以备使用。如果在为类编写自己的构造函数时遵循同样的规则，就会发现，即便是最复杂的类也可以顺利地初始化，并且不会出现任何问题。

# 4.4　修饰符

前面已经遇到许多所谓的修饰符，即应用于类型或成员的关键字。修饰符可以指定方法的可见性，如 public 或 private；还可以指定一个项的本质，如方法是 virtual 或 abstract。C#有许多访问修饰符，下面讨论完整的修饰符列表。

## 4.4.1　访问修饰符

表 4-1 中的修饰符确定了是否允许其他代码访问某一项。

表 4-1

| 修　饰　符 | 应　用　于 | 说　　明 |
|---|---|---|
| public | 所有类型或成员 | 任何代码均可以访问该项 |
| protected | 类型和内嵌类型的所有成员 | 只有派生的类型能访问该项 |
| internal | 所有类型或成员 | 只能在包含它的程序集中访问该项 |
| private | 类型和内嵌类型的所有成员 | 只能在它所属的类型中访问该项 |
| protected internal | 类型和内嵌类型的所有成员 | 只能在包含它的程序集和派生类型的任何代码中访问该项。实际上，这意味着 protected 或 internal |
| private protected | 类型和内嵌类型的所有成员 | 访问修饰符 protected internal 表示 protected 或 internal，与此相反，private protected 将 private 与 protected 组合在一起，表示 private 和 protected。只允许访问同一程序集中的派生类型，而不允许访问其他程序集中的派生类型。这个访问修饰符在 C# 7.2 中是新增的 |

**注意：**
public、protected 和 private 是逻辑访问修饰符。internal 是一个物理访问修饰符，其边界是一个程序集。

注意，类型定义可以是内部或公有的，这取决于是否希望在包含类型的程序集外部访问它：

```
public class MyClass
{
  // ...
```

不能把类型定义为 protected、private 或 protected internal，因为这些修饰符对于包含在名称空间中的类型没有意义。因此这些修饰符只能应用于成员。但是，可以用这些修饰符定义嵌套的类型(即，包含在其他类型中的类型)，因为在这种情况下，类型也具有成员的状态。于是，下面的代码是合法的：

```
public class OuterClass
{
  protected class InnerClass
  {
    // ...
  }
  // ...
}
```

如果有嵌套的类型，则内部的类型总是可以访问外部类型的所有成员。所以，在上面的代码中，InnerClass 中的代码可以访问 OuterClass 的所有成员，甚至可以访问 OuterClass 的私有成员。

### 4.4.2 其他修饰符

表 4-2 中的修饰符可以应用于类型的成员，而且有不同的用途。在应用于类型时，其中的几个修饰符也是有意义的。

表 4-2

| 修 饰 符 | 应 用 于 | 说 明 |
|---|---|---|
| new | 函数成员 | 成员用相同的签名隐藏继承的成员 |
| static | 所有成员 | 成员不作用于类的具体实例，也称为类成员，而不是实例成员 |
| virtual | 仅函数成员 | 成员可以由派生类重写 |
| abstract | 仅函数成员 | 虚拟成员定义了成员的签名，但没有提供实现代码 |
| override | 仅函数成员 | 成员重写了继承的虚拟或抽象成员 |
| sealed | 类、方法和属性 | 对于类，不能继承自密封类。对于属性和方法，成员重写已继承的虚拟成员，但任何派生类中的任何成员都不能重写该成员。该修饰符必须与 override 一起使用 |
| extern | 仅静态[DllImport]方法 | 成员在外部用另一种语言实现。这个关键字的用法参见第 17 章 |

## 4.5 接口

如前所述，如果一个类派生自一个接口，声明这个类就会实现某些函数。并不是所有的面向对象语言都支持接口，所以本节将详细介绍 C#接口的实现。下面列出 Microsoft 预定义的一个接口 System.IDisposable 的完整定义。IDisposable 包含一个方法 Dispose()，该方法由类实现，用于清理代码：

```
public interface IDisposable
{
  void Dispose();
}
```

上面的代码说明，声明接口在语法上与声明抽象类完全相同，但不允许提供接口中任何成员的实现方式。一般情况下，接口只能包含方法、属性、索引器和事件的声明。

比较接口和抽象类：抽象类可以有实现代码或没有实现代码的抽象成员。然而，接口不能有任何实现代码；它是纯粹抽象的。因为接口的成员总是抽象的，所以接口不需要 abstract 关键字。

类似于抽象类，永远不能实例化接口，它只能包含其成员的签名。此外，可以声明接口类型的变量。

接口既不能有构造函数(如何构建不能实例化的对象？)也不能有字段(因为这隐含了某些内部的实现方式)。接口定义也不允许包含运算符重载，但设计语言时总是会讨论这个可能性，未来可能会改变。

在接口定义中还不允许声明成员的修饰符。接口成员总是隐式为 public，不能声明为 virtual。如果需要，就应由实现的类来声明，因此最好实现类来声明访问修饰符，就像本节的代码那样。

例如，IDisposable。如果类希望声明为公有类型，以便它实现方法 Dispose()，该类就必须实现 IDisposable。在 C#中，这表示该类派生自 IDisposable 类。

```
class SomeClass: IDisposable
{
  // This class MUST contain an implementation of the
  // IDisposable.Dispose() method, otherwise
  // you get a compilation error.
  public void Dispose()
  {
    // implementation of Dispose() method
  }
  // rest of class
}
```

在这个例子中，如果 SomeClass 派生自 IDisposable 类，但不包含与 IDisposable 类中签名相同的 Dispose()实现代码，就会得到一个编译错误，因为该类破坏了实现 IDisposable 的一致协定。当然，编译器允许类有一个不派生自 IDisposable 类的 Dispose()方法。问题是其他代码无法识别出 SomeClass 类，来支持 IDisposable 特性。

**注意：**

*IDisposable 是一个相当简单的接口，它只定义了一个方法。大多数接口都包含许多成员。IDisposable 的正确实现代码没有这么简单，参见第 17 章。*

### 4.5.1　定义和实现接口

下面开发一个遵循接口继承规范的小例子来说明如何定义和使用接口。这个例子建立在银行账户的基础上。假定编写代码，最终允许在银行账户之间进行计算机转账业务。许多公司可以实现银行账户，但它们一致认为，表示银行账户的所有类都实现接口 IBankAccount。该接口包含一个用于存取款的方法和一个返回余额的属性。这个接口还允许外部代码识别由不同银行账户实现的各种银行账户类。我们的目的是允许银行账户彼此通信，以便在账户之间进行转账业务，但还没有介绍这个功能。

为了使例子简单一些，我们把本例的所有代码都放在同一个源文件中，但实际上不同的银行账户类不仅会编译到不同的程序集中，而且这些程序集位于不同银行的不同机器上。但这些内容对于我们的目的过于复杂了。为了保留一定的真实性，我们为不同的公司定义不同的名称空间。

首先，需要定义 IBankAccount 接口(代码文件 UsingInterfaces/IBankAccount.cs)：

```
namespace Wrox.ProCSharp
{
  public interface IBankAccount
  {
    void PayIn(decimal amount);
    bool Withdraw(decimal amount);
    decimal Balance { get; }
  }
}
```

注意，接口的名称为 IBankAccount。接口名称通常以字母 I 开头，以便知道这是一个接口。

**注意：**

*如第 2 章所述，在大多数情况下，.NET 的用法规则不鼓励采用所谓的 Hungarian 表示法，在名称的前面加一个字母，表示所定义对象的类型。接口是少数几个推荐使用 Hungarian 表示法的例外之一。*

现在可以编写表示银行账户的类了。这些类不必彼此相关，它们可以是完全不同的类。但它们都表示银行账户，因为它们都实现了 IBankAccount 接口。

下面是第一个类，一个由 Royal Bank of Venus 运行的存款账户(代码文件 UsingInterfaces/VenusBank.cs)：

```
namespace Wrox.ProCSharp.VenusBank
{
  public class SaverAccount: IBankAccount
  {
    private decimal _balance;
```

```
        public void PayIn(decimal amount) => _balance += amount;
        public bool Withdraw(decimal amount)
        {
          if (_balance >= amount)
          {
            _balance -= amount;
            return true;
          }
          Console.WriteLine("Withdrawal attempt failed.");
          return false;
        }
        public decimal Balance => _balance;
        public override string ToString() =>
          $"Venus Bank Saver: Balance = {_balance,6:C}";
      }
    }
```

实现这个类的代码的作用一目了然。其中包含一个私有字段 balance，当存款或取款时就调整这个字段。如果因为账户中的金额不足而取款失败，就会显示一条错误消息。还要注意，因为我们要使代码尽可能简单，所以不实现额外的属性，如账户持有人的姓名。在现实生活中，这是最基本的信息，但对于本例不必要这么复杂。

在这段代码中，唯一有趣的一行是类的声明：

```
public class SaverAccount: IBankAccount
```

SaverAccount 派生自一个接口 IBankAccount，我们没有显式指出任何其他基类(当然这表示 SaverAccount 直接派生自 System.Object)。另外，从接口中派生完全独立于从类中派生。

SaverAccount 派生自 IBankAccount，表示它获得了 IBankAccount 的所有成员，但接口实际上并不实现其方法，所以 SaverAccount 必须提供这些方法的所有实现代码。如果缺少实现代码，编译器就会产生错误。接口仅表示其成员的存在性，类负责确定这些成员是虚拟还是抽象的(但只有在类本身是抽象的，这些函数才能是抽象的)。在本例中，接口的任何函数不必是虚拟的。

为了说明不同的类如何实现相同的接口，下面假定 Planetary Bank of Jupiter 还实现一个类 GoldAccount 来表示其银行账户中的一个(代码文件 UsingInterfaces/JupiterBank.cs )：

```
namespace Wrox.ProCSharp.JupiterBank
{
  public class GoldAccount: IBankAccount
  {
    // ...
  }
}
```

这里没有列出 GoldAccount 类的细节，因为在本例中它基本上与 SaverAccount 的实现代码相同。GoldAccount 与 SaverAccount 没有关系，它们只是碰巧实现相同的接口而已。

有了自己的类后，就可以测试它们了。首先需要一些 using 语句：

```
using Wrox.ProCSharp;
using Wrox.ProCSharp.VenusBank;
using Wrox.ProCSharp.JupiterBank;
```

然后需要一个 Main()方法(代码文件 UsingInterfaces/Program.cs)：

```
namespace Wrox.ProCSharp
{
  class Program
  {
    static void Main()
    {
      IBankAccount venusAccount = new SaverAccount();
      IBankAccount jupiterAccount = new GoldAccount();
      venusAccount.PayIn(200);
      venusAccount.Withdraw(100);
      Console.WriteLine(venusAccount.ToString());
      jupiterAccount.PayIn(500);
      jupiterAccount.Withdraw(600);
      jupiterAccount.Withdraw(100);
      Console.WriteLine(jupiterAccount.ToString());
    }
  }
}
```

这段代码的执行结果如下：

```
> BankAccounts
Venus Bank Saver: Balance = $100.00
Withdrawal attempt failed.
Jupiter Bank Saver: Balance = $400.00
```

在这段代码中，要点是把两个引用变量声明为 IBankAccount 引用的方式。这表示它们可以指向实现这个接口的任何类的任何实例。但我们只能通过这些引用调用接口的一部分方法——如果要调用由类实现的但不在接口中的方法，就需要把引用强制转换为合适的类型。在这段代码中，我们调用了 ToString()(不是 IBankAccount 实现的)，但没有进行任何显式的强制转换，这只是因为 ToString()是一个 System.Object()方法，因此 C#编译器知道任何类都支持这个方法(换言之，从任何接口到 System.Object 的数据类型强制转换是隐式的)。第 6 章将介绍强制转换的语法。

接口引用完全可以看成类引用——但接口引用的强大之处在于，它可以引用任何实现该接口的类。例如，我们可以构造接口数组，其中数组的每个元素都是不同的类：

```
IBankAccount[] accounts = new IBankAccount[2];
accounts[0] = new SaverAccount();
accounts[1] = new GoldAccount();
```

但注意，如果编写了如下代码，就会生成一个编译器错误：

```
accounts[1] = new SomeOtherClass(); // SomeOtherClass does NOT implement
// IBankAccount: WRONG!!
```

这会导致如下所示的编译错误：

```
Cannot implicitly convert type 'Wrox.ProCSharp. SomeOtherClass' to
'Wrox.ProCSharp.IBankAccount'
```

### 4.5.2　派生的接口

接口可以彼此继承，其方式与类的继承方式相同。下面通过定义一个新的 ITransferBankAccount 接口来说明这个概念，该接口的功能与 IBankAccount 相同，只是又定义了一个方法，把资金直接转到另一个账户上(代码文件 UsingInterfaces/ITransferBankAccount )：

```
namespace Wrox.ProCSharp
{
  public interface ITransferBankAccount: IBankAccount
  {
    bool TransferTo(IBankAccount destination, decimal amount);
  }
}
```

因为 ITransferBankAccount 派生自 IBankAccount，所以它拥有 IBankAccount 的所有成员和它自己的成员。这表示实现(派生自)ITransferBankAccount 的任何类都必须实现 IBankAccount 的所有方法和在 ITransferBankAccount 中定义的新方法 TransferTo()。没有实现所有这些方法就会产生一个编译错误。

注意，TransferTo()方法对于目标账户使用了 IBankAccount 接口引用。这说明了接口的用途：在实现并调用这个方法时，不必知道转账的对象类型，只需要知道该对象实现 IBankAccount 即可。

下面说明 ITransferBankAccount：假定 Planetary Bank of Jupiter 还提供了一个当前账户。CurrentAccount 类的大多数实现代码与 SaverAccount 和 GoldAccount 的实现代码相同(这仅是为了使例子更简单，一般是不会这样的)，所以在下面的代码中，我们仅突出显示了不同的地方(代码文件 UsingInterfaces/JupiterBank.cs )：

```
public class CurrentAccount: ITransferBankAccount
{
  private decimal _balance;
  public void PayIn(decimal amount) => _balance += amount;
  public bool Withdraw(decimal amount)
  {
    if (_balance >= amount)
    {
      _balance -= amount;
      return true;
    }
```

```
    Console.WriteLine("Withdrawal attempt failed.");
    return false;
  }

  public decimal Balance => _balance;
  public bool TransferTo(IBankAccount destination, decimal amount)
  {
    bool result = Withdraw(amount);
    if (result)
    {
      destination.PayIn(amount);
    }
    return result;
  }
  public override string ToString() =>
    $"Jupiter Bank Current Account: Balance = {_balance,6:C}";
}
```

可以用下面的代码验证该类：

```
static void Main()
{
  IBankAccount venusAccount = new SaverAccount();
  ITransferBankAccount jupiterAccount = new CurrentAccount();
  venusAccount.PayIn(200);
  jupiterAccount.PayIn(500);
  jupiterAccount.TransferTo(venusAccount, 100);
  Console.WriteLine(venusAccount.ToString());
  Console.WriteLine(jupiterAccount.ToString());
}
```

这段代码的结果如下所示，可以验证，其中说明了正确的转账金额：

```
> CurrentAccount
Venus Bank Saver: Balance = $300.00
Jupiter Bank Current Account: Balance = $400.00
```

# 4.6　is 和 as 运算符

在结束接口和类的继承之前，需要介绍两个与继承有关的重要运算符：is 和 as。

如前所述，可以把具体类型的对象直接分配给基类或接口——如果这些类型在层次结构中有直接关系。例如，前面创建的 SaverAccount 可以直接分配给 IBankAccount，因为 SaverAccount 类型实现了 IBankAccount 接口：

```
IBankAccount venusAccount = new SaverAccount();
```

如果一个方法接受一个对象类型，现在希望访问 IBankAccount 成员，该怎么办？该对象类型没有 IBankAccount 接口的成员。此时可以进行类型转换。把对象(也可以使用任何接口中任意类型的参数，把它转换为需要的类型)转换为 IBankAccount，再处理它：

```
public void WorkWithManyDifferentObjects(object o)
{
  IBankAccount account = (IBankAccount)o;
  // work with the account
}
```

只要总是给这个方法提供一个 IBankAccount 类型的对象，这就是有效的。当然，如果接受一个 object 类型的对象，有时就会传递无效的对象。此时会得到 InvalidCastException 异常。在正常情况下接受异常从来都不好，详见第 14 章。此时应使用 is 和 as 运算符。

不是直接进行类型转换，而应检查参数是否实现了接口 IBankAccount。as 运算符的工作原理类似于类层次结构中的 cast 运算符——它返回对象的引用。然而，它从不抛出 InvalidCastException 异常。相反，如果对象不是所要求的类型，这个运算符就返回 null。这里，最好在使用引用前验证它是否为空，否则以后使用以下引用，就会抛出 NullReferenceException 异常：

```
public void WorkWithManyDifferentObjects(object o)
{
  IBankAccount account = o as IBankAccount;
```

```
  if (account != null)
  {
    // work with the account
  }
}
```

除了使用 as 运算符之外，还可以使用 is 运算符。is 运算符根据条件是否满足，对象是否使用指定的类型，返回 true 或 false。如果条件为 true，则将所得的对象写入声明为匹配类型的变量中，如下面的代码片段所示：

```
public void WorkWithManyDifferentObjects(object o)
{
  if (o is IBankAccount account)
  {
    // work with the account
  }
}
```

**注意：**
向 is 运算符添加变量声明是 C# 7 的一个新特性。这是模式匹配功能的一部分，详见第 13 章。

在类层次结构内部的类型转换不会抛出基于类型转换的异常，且使用 is 和 as 运算符都是可行的。

## 4.7　小结

本章介绍了如何在 C#中进行继承。C#支持多接口继承和单一实现继承，还提供了许多有用的语法结构，以使代码更健壮，如 override 关键字，它表示函数应在何时重写基类函数，new 关键字表示函数在何时隐藏基类函数，构造函数初始化器的硬性规则可以确保构造函数以健壮的方式进行交互操作。

第 5 章介绍了 C#语言的一个重要结构：泛型。

# 第 5 章

# 泛　　型

**本章要点**

- 泛型概述
- 创建泛型类
- 泛型类的特性
- 泛型接口
- 泛型结构
- 泛型方法

**本章源代码下载地址(wrox.com)：**

打开 www.wrox.com 的 Download Code 选项卡可下载本章源代码。源代码也可以在 Generics 目录的 https://github.com/ProfessionalCSharp/ProfessionalCSharp7 中找到。本章代码分为以下几个主要的示例文件：

打开网页 http://www.wrox.com/go/professionalcsharp6，单击 Download Code 选项卡即可下载本章源代码。本章代码分为以下几个主要的示例文件：

- LinkedListObjects
- LinkedListSample
- DocumentManager
- Variance
- GenericMethods
- Specialization

## 5.1 泛型概述

泛型是 C#和.NET 的一个重要概念。泛型不仅是 C#编程语言的一部分，而且与程序集中的 IL(Intermediate Language，中间语言)代码紧密地集成。有了泛型，就可以创建独立于被包含类型的类和方法。我们不必给不同的类型编写功能相同的许多方法或类，只创建一个方法或类即可。

另一个减少代码的选项是使用 Object 类，但使用派生自 Object 类的类型进行传递不是类型安全的。泛型类使用泛型类型，并可以根据需要用特定的类型替换泛型类型。这就保证了类型安全性：如果某个类型不支持泛

型类，编译器就会出现错误。

泛型不仅限于类，本章还将介绍用于接口和方法的泛型。用于委托的泛型参见第 8 章。

泛型不仅存在于 C#中，其他语言中有类似的概念。例如，C++模板就与泛型相似。但是，C++模板和.NET 泛型之间有一个很大的区别。对于 C++模板，在用特定的类型实例化模板时，需要模板的源代码。C++编译器为每个属于特定模板实例的类型生成单独的二进制代码。相反，泛型不仅是 C#语言的一种结构，而且是 CLR(公共语言运行库)定义的。所以，即使泛型类是在 C#中定义的，也可以在 Visual Basic 中用一个特定的类型实例化该泛型。

下面几节介绍泛型的优点和缺点，尤其是：

- 性能
- 类型安全性
- 二进制代码重用
- 代码的扩展
- 命名约定

### 5.1.1　性能

泛型的一个主要优点是性能。第 10 章介绍了 System.Collections 和 System.Collections.Generic 名称空间的泛型和非泛型集合类。对值类型使用非泛型集合类，在把值类型转换为引用类型，和把引用类型转换为值类型时，需要进行装箱和拆箱操作。

> **注意：**
> 装箱和拆箱详见第 6 章，这里仅简要复习一下这些术语。

值类型存储在栈上，引用类型存储在堆上。C#类是引用类型，结构是值类型。.NET 很容易把值类型转换为引用类型，所以可以在需要对象(对象是引用类型)的任意地方使用值类型。例如，int 可以赋予一个对象。从值类型转换为引用类型称为装箱。如果方法需要把一个对象作为参数，同时传递一个值类型，装箱操作就会自动进行。另一方面，装箱的值类型可以使用拆箱操作转换为值类型。在拆箱时，需要使用类型强制转换运算符。

下面的例子显示了 System.Collections 名称空间中的 ArrayList 类。ArrayList 存储对象，Add()方法定义为需要把一个对象作为参数，所以要装箱一个整数类型。在读取 ArrayList 中的值时，要进行拆箱，把对象转换为整数类型。可以使用类型强制转换运算符把 ArrayList 集合的第一个元素赋予变量 i1，在访问 int 类型的变量 i2 的 foreach 语句中，也要使用类型强制转换运算符：

```
var list = new ArrayList();
list.Add(44); // boxing - convert a value type to a reference type
int i1 = (int)list[0]; // unboxing - convert a reference type to
// a value type
foreach (int i2 in list)
{
  Console.WriteLine(i2); // unboxing
}
```

装箱和拆箱操作很容易使用，但性能损失比较大，遍历许多项时尤其如此。

System.Collections.Generic 名称空间中的 List<T>类不使用对象，而是在使用时定义类型。在下面的例子中，List<T>类的泛型类型定义为 int，所以 int 类型在 JIT(Just-In-Time)编译器动态生成的类中使用，不再进行装箱和拆箱操作：

```
var list = new List<int>();
list.Add(44); // no boxing - value types are stored in the List<int>
int i1 = list[0]; // no unboxing, no cast needed
foreach (int i2 in list)
{
  Console.WriteLine(i2);
}
```

### 5.1.2 类型安全

泛型的另一个特性是类型安全。与 ArrayList 类一样，如果使用对象，就可以在这个集合中添加任意类型。下面的例子在 ArrayList 类型的集合中添加一个整数、一个字符串和一个 MyClass 类型的对象：

```
var list = new ArrayList();
list.Add(44);
list.Add("mystring");
list.Add(new MyClass());
```

如果这个集合使用下面的 foreach 语句迭代，而该 foreach 语句使用整数元素来迭代，编译器就会接受这段代码。但并不是集合中的所有元素都可以强制转换为 int，所以会出现一个运行时异常：

```
foreach (int i in list)
{
  Console.WriteLine(i);
}
```

错误应尽早发现。在泛型类 List<T>中，泛型类型 T 定义了允许使用的类型。有了 List<int>的定义，就只能把整数类型添加到集合中。编译器不会编译这段代码，因为 Add()方法的参数无效：

```
var list = new List<int>();
list.Add(44);
list.Add("mystring"); // compile time error
list.Add(new MyClass()); // compile time error
```

### 5.1.3 二进制代码的重用

泛型允许更好地重用二进制代码。泛型类可以定义一次，并且可以用许多不同的类型实例化。不需要像 C++ 模板那样访问源代码。

例如，System.Collections.Generic 名称空间中的 List<T>类用一个 int、一个字符串和一个 MyClass 类型实例化：

```
var list = new List<int>();
list.Add(44);
var stringList = new List<string>();
stringList.Add("mystring");
var myClassList = new List<MyClass>();
myClassList.Add(new MyClass());
```

泛型类型可以在一种语言中定义，在任何其他.NET 语言中使用。

### 5.1.4 代码的扩展

在用不同的特定类型实例化泛型时，会创建多少代码？因为泛型类的定义会放在程序集中，所以用特定类型实例化泛型类不会在 IL 代码中复制这些类。但是，在 JIT 编译器把泛型类编译为本地代码时，会给每个值类型创建一个新类。引用类型共享同一个本地类的所有相同的实现代码。这是因为引用类型在实例化的泛型类中只需要 4 个字节的内存地址(32 位系统)，就可以引用一个引用类型。值类型包含在实例化的泛型类的内存中，同时因为每个值类型对内存的要求都不同，所以要为每个值类型实例化一个新类。

### 5.1.5 命名约定

如果在程序中使用泛型，在区分泛型类型和非泛型类型时就会有一定的帮助。下面是泛型类型的命名规则：

- 泛型类型的名称用字母 T 作为前缀。
- 如果没有特殊的要求，泛型类型允许用任意类替代，且只使用了一个泛型类型，就可以用字符 T 作为泛型类型的名称。

```
public class List<T> { }
public class LinkedList<T> { }
```

- 如果泛型类型有特定的要求(例如，它必须实现一个接口或派生自基类)，或者使用了两个或多个泛型类型，就应给泛型类型使用描述性的名称：

```
public delegate void EventHandler<TEventArgs>(object sender,
  TEventArgs e);
public delegate TOutput Converter<TInput, TOutput>(TInput from);
public class SortedList<TKey, TValue> { }
```

## 5.2 创建泛型类

首先介绍一个一般的、非泛型的简化链表类，它可以包含任意类型的对象，以后再把这个类转化为泛型类。

在链表中，一个元素引用下一个元素。所以必须创建一个类，它将对象封装在链表中，并引用下一个对象。类 LinkedListNode 包含一个属性 Value，该属性用构造函数初始化。另外，LinkedListNode 类包含对链表中下一个元素和上一个元素的引用，这些元素都可以从属性中访问(代码文件 LinkedListObjects/LinkedListNode.cs)。

```
public class LinkedListNode
{
  public LinkedListNode(object value) => Value = value;

  public object Value { get; }
  public LinkedListNode Next { get; internal set; }
  public LinkedListNode Prev { get; internal set; }
}
```

LinkedList 类包含 LinkedListNode 类型的 First 和 Last 属性，它们分别标记了链表的头尾。AddLast()方法在链表尾添加一个新元素。首先创建一个 LinkedListNode 类型的对象。如果链表是空的，First 和 Last 属性就设置为该新元素；否则，就把新元素添加为链表中的最后一个元素。通过实现 GetEnumerator()方法，可以用 foreach 语句遍历链表。GetEnumerator()方法使用 yield 语句创建一个枚举器类型。

```
public class LinkedList: IEnumerable
{
  public LinkedListNode First { get; private set; }
  public LinkedListNode Last { get; private set; }
  public LinkedListNode AddLast(object node)
  {
    var newNode = new LinkedListNode(node);
    if (First == null)
    {
      First = newNode;
      Last = First;
    }
    else
    {
      LinkedListNode previous = Last;
      Last.Next = newNode;
      Last = newNode;
      Last.Prev = previous;
    }
    return newNode;
  }

  public IEnumerator GetEnumerator()
  {
    LinkedListNode current = First;
    while (current != null)
    {
      yield return current.Value;
      current = current.Next;
    }
  }
}
```

**注意：**
yield 语句创建一个枚举器的状态机，详细介绍请参见第 7 章。

现在可以对于任意类型使用 LinkedList 类。在下面的代码段中，实例化了一个新 LinkedList 对象，添加了两

个整数类型和一个字符串类型。整数类型要转换为一个对象，所以执行装箱操作，如前面所述。通过 foreach 语句执行拆箱操作。在 foreach 语句中，链表中的元素被强制转换为整数，所以对于链表中的第 3 个元素，会发生一个运行时异常，因为把它强制转换为 int 时会失败(代码文件 LinkedListObjects/Program.cs)。

```
var list1 = new LinkedList();
list1.AddLast(2);
list1.AddLast(4);
list1.AddLast("6");
foreach (int i in list1)
{
  Console.WriteLine(i);
}
```

下面创建链表的泛型版本。泛型类的定义与一般类类似，只是要使用泛型类型声明。之后，泛型类型就可以在类中用作一个字段成员，或者方法的参数类型。LinkedListNode 类用一个泛型类型 T 声明。属性 Value 的类型是 T，而不是 object。构造函数也变为可以接受 T 类型的对象。也可以返回和设置泛型类型，所以属性 Next 和 Prev 的类型是 LinkedListNode<T>(代码文件 LinkedListSample/LinkedListNode.cs)。

```
public class LinkedListNode<T>
{
  public LinkedListNode(T value) => Value = value;

  public T Value { get; }
  public LinkedListNode<T> Next { get; internal set; }
  public LinkedListNode<T> Prev { get; internal set; }
}
```

下面的代码把 LinkedList 类也改为泛型类。LinkedList<T>包含 LinkedListNode<T>元素。LinkedList 中的类型 T 定义了类型 T 的属性 First 和 Last。AddLast()方法现在接受类型 T 的参数，并实例化 LinkedListNode<T>类型的对象。

除了 IEnumerable 接口，还有一个泛型版本 IEnumerable<T>。IEnumerable<T>派生自 IEnumerable，添加了返回 IEnumerator<T>的 GetEnumerator()方法，LinkedList<T>实现泛型接口 IEnumerable<T>(代码文件 LinkedListSample/LinkedList.cs)。

> **注意：**
> 枚举与接口 IEnumerable 和 IEnumerator 详见第 7 章。

```
public class LinkedList<T>: IEnumerable<T>
{
  public LinkedListNode<T> First { get; private set; }
  public LinkedListNode<T> Last { get; private set; }
  public LinkedListNode<T> AddLast(T node)
  {
    var newNode = new LinkedListNode<T>(node);
    if (First == null)
    {
      First = newNode;
      Last = First;
    }
    else
    {
      LinkedListNode<T> previous = Last;
      Last.Next = newNode;
      Last = newNode;
      Last.Prev = previous;
    }
    return newNode;
  }

  public IEnumerator<T> GetEnumerator()
  {
    LinkedListNode<T> current = First;
    while (current != null)
    {
      yield return current.Value;
      current = current.Next;
    }
  }
```

```
    IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
  }
```

使用泛型 LinkedList<T>，可以用 int 类型实例化它，且不需要装箱操作。如果不使用 AddLast()方法传递 int，就会出现一个编译器错误。使用泛型 IEnumerable<T>，foreach 语句也是类型安全的，如果 foreach 语句中的变量不是 int，就会出现一个编译器错误(代码文件 LinkedListSample/Program.cs)。

```
var list2 = new LinkedList<int>();
list2.AddLast(1);
list2.AddLast(3);
list2.AddLast(5);
foreach (int i in list2)
{
  Console.WriteLine(i);
}
```

同样，可以对于字符串类型使用泛型 LinkedList<T>，将字符串传递给 AddLast()方法。

```
var list3 = new LinkedList<string>();
list3.AddLast("2");
list3.AddLast("four");
list3.AddLast("foo");
foreach (string s in list3)
{
  Console.WriteLine(s);
}
```

**注意：**

*每个处理对象类型的类都可以有泛型实现方式。另外，如果类使用了层次结构，泛型就非常有助于消除不必要的类型强制转换操作。*

## 5.3　泛型类的功能

在创建泛型类时，还需要一些其他 C#关键字。例如，不能把 null 赋予泛型类型。此时，如下一节所述，可以使用 default 关键字。如果泛型类型不需要 Object 类的功能，但需要调用泛型类上的某些特定方法，就可以定义约束。

本节讨论如下主题：

- 默认值
- 约束
- 继承
- 静态成员

首先介绍一个使用泛型文档管理器的示例。文档管理器用于从队列中读写文档。先创建一个新的控制台项目 DocumentManager，并添加 DocumentManager<T>类。AddDocument()方法将一个文档添加到队列中。如果队列不为空，IsDocumentAvailable 只读属性就返回 true(代码文件 DocumentManager/DocumentManager.cs)。

```
using System;
using System.Collections.Generic;
namespace Wrox.ProCSharp.Generics
{
  public class DocumentManager<T>
  {
    private readonly Queue<T> _documentQueue = new Queue<T>();
    private readonly object _lockQueue = new object();

    public void AddDocument(T doc)
    {
      lock (_lockQueue)
      {
        _documentQueue.Enqueue(doc);
      }
    }
```

```
    public bool IsDocumentAvailable => _documentQueue.Count > 0;
  }
}
```

第 21 章将讨论线程和 lock 语句。

### 5.3.1 默认值

现在给 DocumentManager<T>类添加一个 GetDocument()方法。在这个方法中，应把类型 T 指定为 null。但是，不能把 null 赋予泛型类型。原因是泛型类型也可以实例化为值类型，而 null 只能用于引用类型。为了解决这个问题，可以使用 default 关键字。通过 default 关键字，将 null 赋予引用类型，将 0 赋予值类型。

```
public T GetDocument()
{
  T doc = default;
  lock (_lockQueue)
  {
    doc = _documentQueue.Dequeue();
  }
  return doc;
}
```

**注意：**

default 关键字根据上下文可以有多种含义。switch 语句使用 default 定义默认情况。在泛型中，取决于泛型类型是引用类型还是值类型，泛型 default 将泛型类型初始化为 null 或 0。

### 5.3.2 约束

如果泛型类需要调用泛型类型中的方法，就必须添加约束。

对于 DocumentManager<T>，文档的所有标题应在 DisplayAllDocuments()方法中显示。Document 类实现带有 Title 和 Content 只读属性的 IDocument 接口(代码文件 DocumentManager/Document.cs)：

```
public interface IDocument
{
  string Title { get; }
  string Content { get; }
}

public class Document: IDocument
{
  public Document(string title, string content)
  {
    Title = title;
    Content = content;
  }

  public string Title { get; }
  public string Content { get; }
}
```

要使用 DocumentManager<T>类显示文档，可以将类型 T 强制转换为 IDocument 接口，以显示标题：

```
public void DisplayAllDocuments()
{
  foreach (T doc in documentQueue)
  {
    Console.WriteLine(((IDocument)doc).Title);
  }
}
```

问题是，如果类型 T 没有实现 IDocument 接口，这个类型强制转换就会导致一个运行时异常。最好给 DocumentManager<TDocument>类定义一个约束：TDocument 类型必须实现 IDocument 接口。为了在泛型类型的名称中指定该要求，将 T 改为 TDocument。where 子句指定了实现 IDocument 接口的要求(代码文件 DocumentManager/DocumentManager.cs)：

```
public class DocumentManager<TDocument>
  where TDocument: IDocument
{
```

**注意：**

给泛型类型添加约束时，最好包含泛型参数名称的一些信息。现在，示例代码给泛型参数使用 TDocument 来代替 T。对于编译器而言，参数名不重要，但它更具可读性。

这样就可以编写 foreach 语句，从而使类型 TDocument 包含属性 Title。Visual Studio IntelliSense 和编译器都会提供这个支持。

```
public void DisplayAllDocuments()
{
  foreach (TDocument doc in documentQueue)
  {
    Console.WriteLine(doc.Title);
  }
}
```

在 Main()方法中，用 Document 类型实例化 DocumentManager<TDocument>类，而 Document 类型实现了需要的 IDocument 接口。接着添加和显示新文档，检索其中一个文档(代码文件 DocumentManager/Program.cs)：

```
public static void Main()
{
  var dm = new DocumentManager<Document>();
  dm.AddDocument(new Document("Title A", "Sample A"));
  dm.AddDocument(new Document("Title B", "Sample B"));
  dm.DisplayAllDocuments();
  if (dm.IsDocumentAvailable)
  {
    Document d = dm.GetDocument();
    Console.WriteLine(d.Content);
  }
}
```

DocumentManager 现在可以处理任何实现了 IDocument 接口的类。

在示例应用程序中介绍了接口约束。泛型支持几种约束类型，如表 5-1 所示。

表 5-1

| 约　束 | 说　明 |
|---|---|
| where T: struct | 对于结构约束，类型 T 必须是值类型 |
| where T: class | 类约束指定类型 T 必须是引用类型 |
| where T: IFoo | 指定类型 T 必须实现接口 IFoo |
| where T: Foo | 指定类型 T 必须派生自基类 Foo |
| where T: new() | 这是一个构造函数约束，指定类型 T 必须有一个默认构造函数 |
| where T1: T2 | 这个约束也可以指定，类型 T1 派生自泛型类型 T2 |

**注意：**

只能为默认构造函数定义构造函数约束，不能为其他构造函数定义构造函数约束。

使用泛型类型还可以合并多个约束。where T: IFoo, new()约束和 MyClass<T>声明指定，类型 T 必须实现 IFoo 接口，且必须有一个默认构造函数。

```
public class MyClass<T>
  where T: IFoo, new()
{
  //...
```

**注意：**

在 C#中，where 子句的一个重要限制是，不能定义必须由泛型类型实现的运算符。运算符不能在接口中定义。在 where 子句中，只能定义基类、接口和默认构造函数。

### 5.3.3 继承

前面创建的 LinkedList<T>类实现了 IEnumerable<T>接口：

```
public class LinkedList<T>: IEnumerable<T>
{
  //...
```

泛型类型可以实现泛型接口，也可以派生自一个类。泛型类可以派生自泛型基类：

```
public class Base<T>
{
}

public class Derived<T>: Base<T>
{
}
```

其要求是必须重复接口的泛型类型，或者必须指定基类的类型，如下例所示：

```
public class Base<T>
{
}

public class Derived<T>: Base<string>
{
}
```

于是，派生类可以是泛型类或非泛型类。例如，可以定义一个抽象的泛型基类，它在派生类中用一个具体的类实现。这允许对特定类型执行特殊的操作：

```
public abstract class Calc<T>
{
  public abstract T Add(T x, T y);
  public abstract T Sub(T x, T y);
}

public class IntCalc: Calc<int>
{
  public override int Add(int x, int y) => x + y;
  public override int Sub(int x, int y) => x - y;
}
```

还可以创建一个部分的特殊操作，如从 Query 中派生 StringQuery 类，只定义一个泛型参数，如字符串 TResult。要实例化 StringQuery，只需要提供 TRequest 的类型：

```
public class Query<TRequest, TResult>
{
}

public StringQuery<TRequest> : Query<TRequest, string>
{
}
```

### 5.3.4 静态成员

泛型类的静态成员需要特别关注。泛型类的静态成员只能在类的一个实例中共享。下面看一个例子，其中 StaticDemo<T>类包含静态字段 x：

```
public class StaticDemo<T>
{
  public static int x;
}
```

由于同时对一个 string 类型和一个 int 类型使用了 StaticDemo<T>类，因此存在两组静态字段：

```
StaticDemo<string>.x = 4;
```

```
StaticDemo<int>.x = 5;
Console.WriteLine(StaticDemo<string>.x); // writes 4
```

# 5.4 泛型接口

使用泛型可以定义接口，在接口中定义的方法可以带泛型参数。在链表的示例中，实现了 IEnumerable<out T>接口，它定义了 GetEnumerator()方法，以返回 IEnumerator<out T>。.NET 为不同的情况提供了许多泛型接口，例如，IComparable<T>、ICollection<T>和 IExtensibleObject<T>。同一个接口常常存在比较老的非泛型版本，例如，.NET 1.0 有基于对象的 IComparable 接口。IComparable<in T>基于一个泛型类型：

```
public interface IComparable<in T>
{
  int CompareTo(T other);
}
```

**注意：**
不要混淆用于泛型参数的 in 和 out 关键字。参见 5.4.1 节“协变和抗变”。

比较老的非泛型接口 IComparable 需要一个带 CompareTo()方法的对象。这需要强制转换为特定的类型，例如，Person 类要使用 LastName 属性，就需要使用 CompareTo()方法：

```
public class Person: IComparable
{
  public int CompareTo(object obj)
  {
    Person other = obj as Person;
    return this.lastname.CompareTo(other.LastName);
  }
  //
```

实现泛型版本时，不再需要将 object 的类型强制转换为 Person：

```
public class Person: IComparable<Person>
{
  public int CompareTo(Person other) => LastName.CompareTo(other.LastName);

  //...
```

## 5.4.1 协变和抗变

在.NET 4 之前，泛型接口是不变的。.NET 4 通过协变和抗变为泛型接口和泛型委托添加了一个重要的扩展。协变和抗变指对参数和返回值的类型进行转换。例如，可以给一个需要 Shape 参数的方法传送 Rectangle 参数吗？下面用示例说明这些扩展的优点。

在.NET 中，参数类型是抗变的。假定有 Shape 和 Rectangle 类，Rectangle 类派生自 Shape 基类。声明 Display()方法是为了接受 Shape 类型的对象作为其参数：

```
public void Display(Shape o) { }
```

现在可以传递派生自 Shape 基类的任意对象。因为 Rectangle 派生自 Shape，所以 Rectangle 满足 Shape 的所有要求，编译器接受这个方法调用：

```
var r = new Rectangle { Width= 5, Height=2.5 };
Display(r);
```

方法的返回类型是协变的。当方法返回一个 Shape 时，不能把它赋予 Rectangle，因为 Shape 不一定总是 Rectangle。反过来是可行的：如果一个方法像 GetRectangle()方法那样返回一个 Rectangle，

```
public Rectangle GetRectangle();
```

就可以把结果赋予某个 Shape：

```
Shape s = GetRectangle();
```

在.NET Framework 4 版本之前，这种行为方式不适用于泛型。自 C# 4 以后，扩展后的语言支持泛型接口和

泛型委托的协变和抗变。下面开始定义 Shape 基类和 Rectangle 类(代码文件 Variance/Shape.cs 和 Rectangle.cs)：

```
public class Shape
{
  public double Width { get; set; }
  public double Height { get; set; }
  public override string ToString() => $"Width: {Width}, Height: {Height}";
}

public class Rectangle: Shape
{
}
```

### 5.4.2 泛型接口的协变

如果泛型类型用 out 关键字标注，泛型接口就是协变的。这也意味着返回类型只能是 T。接口 IIndex 与类型 T 是协变的，并从一个只读索引器中返回这个类型(代码文件 Variance/IIndex.cs)：

```
public interface IIndex<out T>
{
  T this[int index] { get; }
  int Count { get; }
}
```

IIndex<T>接口用 RectangleCollection 类来实现。RectangleCollection 类为泛型类型 T 定义了 Rectangle：

**注意：**

如果对接口 IIndex 使用了读写索引器，就把泛型类型 T 传递给方法，并从方法中检索这个类型。这不能通过协变来实现——泛型类型必须定义为不变的。不使用 out 和 in 标注，就可以把类型定义为不变的(代码文件 Variance/RectangleCollection.cs)。

```
public class RectangleCollection: IIndex<Rectangle>
{
  private Rectangle[] data = new Rectangle[3]
  {
    new Rectangle { Height=2, Width=5 },
    new Rectangle { Height=3, Width=7 },
    new Rectangle { Height=4.5, Width=2.9 }
  };

  private static RectangleCollection _coll;
  public static RectangleCollection GetRectangles() =>
    _coll ?? (_coll = new RectangleCollection());

  public Rectangle this[int index]
  {
    get
    {
      if (index < 0 || index > data.Length)
        throw new ArgumentOutOfRangeException(nameof(index));
      return data[index];
    }
  }

  public int Count => data.Length;
}
```

**注意：**

RectangleCollection.GetRectangles()方法使用了本章后面将会介绍的合并运算符(coalescing operator)。如果变量 coll1 为 null，则会调用运算符的右侧，以创建 RectangleCollection 的一个新实例，并将其赋给变量 coll1。之后，会从 GetRectangles()方法中返回变量 coll1。这个运算符详见第 6 章。

RectangleCollection.GetRectangle()方法返回一个实现 IIndex<Rectangle>接口的 RectangleCollection 类，所以可以把返回值赋予 IIndex<Rectangle>类型的变量 rectangle。因为接口是协变的，所以也可以把返回值赋予 IIndex<Shape>类型的变量。Shape 不需要 Rectangle 没有提供的内容。使用 shapes 变量，就可以在 for 循环中使用

接口中的索引器和 Count 属性(代码文件 Variance/Program.cs)：

```
public static void Main()
{
  IIndex<Rectangle> rectangles = RectangleCollection.GetRectangles();
  IIndex<Shape> shapes = rectangles;
  for (int i = 0; i < shapes.Count; i++)
  {
    Console.WriteLine(shapes[i]);
  }
}
```

### 5.4.3　泛型接口的抗变

如果泛型类型用 in 关键字标注，泛型接口就是抗变的。这样，接口只能把泛型类型 T 用作其方法的输入(代码文件 Variance/IDisplay.cs)：

```
public interface IDisplay<in T>
{
  void Show(T item);
}
```

ShapeDisplay 类实现 IDisplay<Shape>，并使用 Shape 对象作为输入参数(代码文件 Variance/ShapeDisplay.cs)：

```
public class ShapeDisplay: IDisplay<Shape>
{
  public void Show(Shape s) =>
    Console.WriteLine(
      $"{s.GetType().Name} Width: {s.Width}, Height: {s.Height}");
}
```

创建 ShapeDisplay 的一个新实例，会返回 IDisplay<Shape>，并把它赋予 shapeDisplay 变量。因为 IDisplay<T>是抗变的，所以可以把结果赋予 IDisplay<Rectangle>，其中 Rectangle 派生自 Shape。这次接口的方法只能把泛型类型定义为输入，而 Rectangle 满足 Shape 的所有要求(代码文件 Variance/Program.cs)：

```
public static void Main()
{
  //...
  IDisplay<Shape> shapeDisplay = new ShapeDisplay();
  IDisplay<Rectangle> rectangleDisplay = shapeDisplay;
  rectangleDisplay.Show(rectangles[0]);
}
```

## 5.5　泛型结构

与类相似，结构也可以是泛型的。它们非常类似于泛型类，只是没有继承特性。本节介绍泛型结构 Nullable<T>，它由.NET Framework 定义。

.NET Framework 中的一个泛型结构是 Nullable<T>。数据库中的数字和编程语言中的数字有显著不同的特征，因为数据库中的数字可以为空，而 C#中的数字不能为空。Int32 是一个结构，而结构实现同值类型，所以结构不能为空。这种区别常常令人很头痛，映射数据也要多做许多辅助工作。这个问题不仅存在于数据库中，也存在于把 XML 数据映射到.NET 类型。

一种解决方案是把数据库和 XML 文件中的数字映射为引用类型，因为引用类型可以为空值。但这也会在运行期间带来额外的系统开销。

使用 Nullable<T>结构很容易解决这个问题。下面的代码段说明了如何定义 Nullable<T>的一个简化版本。结构 Nullable<T>定义了一个约束：其中的泛型类型 T 必须是一个结构。把类定义为泛型类型后，就没有低系统开销这个优点了，而且因为类的对象可以为空，所以对类使用 Nullable<T>类型是没有意义的。除了 Nullable<T>定义的 T 类型之外，唯一的系统开销是 hasValue 布尔字段，它确定是设置对应的值，还是使之为空。除此之外，泛型结构还定义了只读属性 HasValue 和 Value，以及一些运算符重载。把 Nullable<T>类型强制转换为 T 类型的运算符重载是显式定义的，因为当 hasValue 为 false 时，它会抛出一个异常。强制转换为 Nullable<T>类型的运算符重载定义为隐式的，因为它总是能成功地转换：

```
public struct Nullable<T>
  where T: struct
{
  public Nullable(T value)
  {
    _hasValue = true;
    _value = value;
  }

  private bool _hasValue;
  public bool HasValue => _hasValue;

  private T _value;
  public T Value
  {
    get
    {
      if (!_hasValue)
      {
        throw new InvalidOperationException("no value");
      }
      return _value;
    }
  }

  public static explicit operator T(Nullable<T> value) => _value.Value;

  public static implicit operator Nullable<T>(T value) =>
    new Nullable<T>(value);

  public override string ToString() => !HasValue ? string.Empty :
    _value.ToString();
}
```

在这个例子中，Nullable<T>用 Nullable<int>实例化。变量 x 现在可以用作一个 int，进行赋值或使用运算符执行一些计算。这是因为强制转换了 Nullable<T>类型的运算符。但是，x 还可以为空。Nullable<T>的 HasValue 和 Value 属性可以检查是否有一个值，该值是否可以访问：

```
Nullable<int> x;
x = 4;
x += 3;
if (x.HasValue)
{
  int y = x.Value;
}
x = null;
```

因为可空类型使用得非常频繁，所以 C#有一种特殊的语法，它用于定义可空类型的变量。定义这类变量时，不使用泛型结构的语法，而使用“?”运算符。在下面的例子中，变量 x1 和 x2 都是可空的 int 类型的实例：

```
Nullable<int> x1;
int? x2;
```

可空类型可以与 null 和数字比较，如上所示。这里，x 的值与 null 比较，如果 x 不是 null，它就与小于 0 的值比较：

```
int? x = GetNullableType();
if (x == null)
{
  Console.WriteLine("x is null");
}
else if (x < 0)
{
  Console.WriteLine("x is smaller than 0");
}
```

知道了 Nullable<T>是如何定义的之后，下面就使用可空类型。可空类型还可以与算术运算符一起使用。变量 x3 是变量 x1 和 x2 的和。如果这两个可空变量中任何一个的值是 null，它们的和就是 null。

```
int? x1 = GetNullableType();
int? x2 = GetNullableType();
int? x3 = x1 + x2;
```

**注意：**

这里调用的 GetNullableType()方法只是一个占位符，它对于任何方法都返回一个可空的 int。为了进行测试，简单起见，可以使实现的 GetNullableType()返回 null 或返回任意整数。

非可空类型可以转换为可空类型。从非可空类型转换为可空类型时，在不需要强制类型转换的地方可以进行隐式转换。这种转换总是成功的：

```
int y1 = 4;
int? x1 = y1;
```

但从可空类型转换为非可空类型可能会失败。如果可空类型的值是 null，并且把 null 值赋予非可空类型，就会抛出 InvalidOperationException 类型的异常。这就是需要类型强制转换运算符进行显式转换的原因：

```
int? x1 = GetNullableType();
int y1 = (int)x1;
```

如果不进行显式类型转换，还可以使用合并运算符从可空类型转换为非可空类型。合并运算符的语法是“??”，为转换定义了一个默认值，以防可空类型的值是 null。这里，如果 x1 是 null，y1 的值就是 0。

```
int? x1 = GetNullableType();
int y1 = x1 ?? 0;
```

# 5.6　泛型方法

除了定义泛型类之外，还可以定义泛型方法。在泛型方法中，泛型类型用方法声明来定义。泛型方法可以在非泛型类中定义。

Swap<T>()方法把 T 定义为泛型类型，该泛型类型用于两个参数和一个变量 temp：

```
void Swap<T>(ref T x, ref T y)
{
  T temp;
  temp = x;
  x = y;
  y = temp;
}
```

把泛型类型赋予方法调用，就可以调用泛型方法：

```
int i = 4;
int j = 5;
Swap<int>(ref i, ref j);
```

但是，因为 C#编译器会通过调用 Swap()方法来获取参数的类型，所以不需要把泛型类型赋予方法调用。泛型方法可以像非泛型方法那样调用：

```
int i = 4;
int j = 5;
Swap(ref i, ref j);
```

## 5.6.1　泛型方法示例

下面的例子使用泛型方法累加集合中的所有元素。为了说明泛型方法的功能，下面使用包含 Name 和 Balance 属性的 Account 类(代码文件 GenericMethods/Account.cs)：

```
public class Account
{
  public string Name { get; }
  public decimal Balance { get; }
  public Account(string name, Decimal balance)
  {
    Name = name;
    Balance = balance;
  }
}
```

其中应累加余额的所有账户操作都添加到 List<Account>类型的账户列表中(代码文件 GenericMethods/Program.cs)：

```
var accounts = new List<Account>()
{
  new Account("Christian", 1500),
  new Account("Stephanie", 2200),
  new Account("Angela", 1800),
  new Account("Matthias", 2400),
  new Account("Katharina", 3800),
};
```

累加所有 Account 对象的传统方式是用 foreach 语句遍历所有的 Account 对象，如下所示。foreach 语句使用 IEnumerable 接口迭代集合的元素，所以 AccumulateSimple()方法的参数是 IEnumerable 类型。foreach 语句处理实现 IEnumerable 接口的每个对象。这样，AccumulateSimple()方法就可以用于所有实现 IEnumerable<Account>接口的集合类。在这个方法的实现代码中，直接访问 Account 对象的 Balance 属性(代码文件 GenericMethods/Algorithms.cs)：

```
public static class Algorithms
{
  public static decimal AccumulateSimple(IEnumerable<Account> source)
  {
    decimal sum = 0;
    foreach (Account a in source)
    {
      sum += a.Balance;
    }
    return sum;
  }
}
```

AccumulateSimple()方法的调用方式如下：

```
decimal amount = Algorithms.AccumulateSimple(accounts);
```

## 5.6.2 带约束的泛型方法

第一个实现代码的问题是，它只能用于 Account 对象。使用泛型方法就可以避免这个问题。

Accumulate()方法的第二个版本接受实现了 IAccount 接口的任意类型。如前面的泛型类所述，泛型类型可以用 where 子句来限制。用于泛型类的这个子句也可以用于泛型方法。Accumulate()方法的参数改为 IEnumerable<T>。IEnumerable<T>是泛型集合类实现的泛型接口(代码文件 GenericMethods/Algorithms.cs)。

```
public static decimal Accumulate<TAccount>(IEnumerable<TAccount> source)
  where TAccount: IAccount
{
  decimal sum = 0;
  foreach (TAccount a in source)
  {
    sum += a.Balance;
  }
  return sum;
}
```

重构的 Account 类现在实现接口 IAccount(代码文件 GenericMethods/Account.cs)：

```
public class Account: IAccount
{
  //...
```

IAccount 接口定义了只读属性 Balance 和 Name(代码文件 GenericMethods/IAccount.cs)：

```
public interface IAccount
{
  decimal Balance { get; }
  string Name { get; }
}
```

将 Account 类型定义为泛型类型参数，就可以调用新的 Accumulate()方法(代码文件 GenericMethods/Program.cs)：

```
decimal amount = Algorithm.Accumulate<Account>(accounts);
```

因为编译器会从方法的参数类型中自动推断出泛型类型参数，所以以如下方式调用 Accumulate()方法是有效的：

```
decimal amount = Algorithm.Accumulate(accounts);
```

### 5.6.3 带委托的泛型方法

泛型类型实现 IAccount 接口的要求过于严格。下面的示例提示了，如何通过传递一个泛型委托来修改 Accumulate()方法。第 8 章详细介绍了如何使用泛型委托，以及如何使用 lambda 表达式。

这个 Accumulate()方法使用两个泛型参数 T1 和 T2。第一个参数 T1 用于实现 IEnumerable<T1>参数的集合，第二个参数使用泛型委托 Func<T1, T2, TResult>。其中，第 2 个和第 3 个泛型参数都是 T2 类型。需要传递的方法有两个输入参数(T1 和 T2)和一个 T2 类型的返回值(代码文件 GenericMethods/Algorithms.cs)：

```
public static T2 Accumulate<T1, T2>(IEnumerable<T1> source,
  Func<T1, T2, T2> action)
{
  T2 sum = default(T2);
  foreach (T1 item in source)
  {
    sum = action(item, sum);
  }
  return sum;
}
```

在调用这个方法时，需要指定泛型参数类型，因为编译器不能自动推断出该类型。对于方法的第 1 个参数，所赋予的 accounts 集合是 IEnumerable<Account>类型。对于第 2 个参数，使用一个 lambda 表达式来定义 Account 和 decimal 类型的两个参数，返回一个小数。对于每一项，通过 Accumulate()方法调用这个 lambda 表达式(代码文件 GenericMethods/Program.cs)：

```
decimal amount = Algorithm.Accumulate<Account, decimal>(
accounts, (item, sum) => sum += item.Balance);
```

不要为这种语法伤脑筋。该示例仅说明了扩展 Accumulate()方法的可能方式。lambda 表达式详见第 8 章。

### 5.6.4 泛型方法规范

泛型方法可以重载，为特定的类型定义规范。这也适用于带泛型参数的方法。Foo()方法定义了 4 个版本，第 1 个版本接受一个泛型参数，第 2 个版本是用于 int 参数的专用版本。第 3 个 Foo 方法接受两个泛型参数，第 4 个版本是第 3 个版本的专用版本，其第一个参数是 int 类型。在编译期间，会使用最佳匹配。如果传递了一个 int，就选择带 int 参数的方法。对于任何其他参数类型，编译器会选择方法的泛型版本(代码文件 Specialization/Program.cs)：

```
public class MethodOverloads
{
  public void Foo<T>(T obj) =>
    Console.WriteLine($"Foo<T>(T obj), obj type: {obj.GetType().Name}");

  public void Foo(int x) =>
    Console.WriteLine("Foo(int x)");

  public void Foo<T1, T2>(T1 obj1, T2 obj2) =>
    Console.WriteLine($"Foo<T1, T2>(T1 obj1, T2 obj2); " +
      $"{obj1.GetType().Name} {obj2.GetType().Name}");

  public void Foo<T>(int obj1, T obj2) =>
    Console.WriteLine($"Foo<T>(int obj1, T obj2); {obj2.GetType().Name}");

  public void Bar<T>(T obj) => Foo(obj);
}
```

Foo()方法现在可以通过任意参数类型来调用。下面的示例代码传递了 int 和 string 值，调用所有 4 个 Foo 方法：

```
static void Main()
```

```
{
  var test = new MethodOverloads();
  test.Foo(33);
  test.Foo("abc");
  test.Foo("abc", 42);
  test.Foo(33, "abc");
}
```

运行该程序，可以从输出中看出选择了最佳匹配的方法：

```
Foo(int x)
Foo<T>(T obj), obj type: String
Foo<T1, T2>(T1 obj1, T2 obj2); String Int32
Foo<T>(int obj1, T obj2); String
```

需要注意的是，所调用的方法是在编译期间而不是运行期间定义的。这很容易举例说明：添加一个调用 Foo()方法的 Bar()泛型方法，并传递泛型参数值：

```
public class MethodOverloads
{
  // ...
  public void Bar<T>(T obj) =>
    Foo(obj);
```

Main()方法现在改为调用传递一个 int 值的 Bar()方法：

```
static void Main()
{
  var test = new MethodOverloads();
  test.Bar(44);
```

从控制台的输出可以看出，Bar()方法选择了泛型 Foo()方法，而不是选择用 int 参数重载的 Foo()方法。原因是编译器是在编译期间选择 Bar()方法调用的 Foo()方法。由于 Bar()方法定义了一个泛型参数，而且泛型 Foo()方法匹配这个类型，因此调用了 Foo()方法。在运行期间给 Bar()方法传递一个 int 值不会改变这一点。

```
Foo<T>(T obj), obj type: Int32
```

## 5.7 小结

本章介绍了 CLR 中一个非常重要的特性：泛型。通过泛型类可以创建独立于类型的类，泛型方法是独立于类型的方法。接口、结构和委托也可以用泛型的方式创建。泛型引入了一种新的编程方式。我们介绍了如何实现相应的算法(尤其是操作和谓词)以用于不同的类，而且它们都是类型安全的。泛型委托可以去除集合中的算法。

本书还将探讨泛型的更多特性和用法。第 8 章介绍了常常实现为泛型的委托，第 10 章论述了泛型集合类，第 12 章讨论了泛型扩展方法。第 6 章说明了运算符和强制类型转换。

# 第6章 运算符和类型强制转换

**本章要点**

- C#中的运算符
- 使用 nameof 运算符和空值条件运算符
- 隐式和显式转换
- 使用装箱技术把值类型转换为引用类型
- 比较值类型和引用类型
- 重载标准的运算符以支持自定义类型
- 实现索引运算符
- 通过类型强制转换在引用类型之间转换

**本章源代码下载地址(wrox.com)：**

打开 www.wrox.com 的 Download Code 选项卡可下载本章源代码。源代码也可以在 OperatorsAndCasts 目录的 https://github.com/ProfessionalCSharp/ProfessionalCSharp7 中找到。本章代码分为以下几个主要的示例文件：

- OperatorsSample
- BinaryCalculations
- OperatorOverloadingSample
- OperatorOverloadingSample2
- OverloadingComparisonSample
- CustomIndexerSample
- CastingSample

## 6.1 运算符和类型转换

前几章介绍了使用 C#编写有用程序所需要的大部分知识。本章将首先讨论基本语言元素，接着论述 C#语言的强大扩展功能。

本章介绍使用运算符的内容，包括 C# 6 添加的运算符，例如空值条件运算符和 nameof 运算符，以及 C# 7 的运算符扩展，例如 is 运算符的模式匹配。本章后面将讨论运算符的重载。本章还会解释如何使用运算符实现

定制功能。

## 6.2　运算符

C#运算符非常类似于 C++和 Java 运算符，但有一些区别。

C#支持表 6-1 中的运算符。

表 6-1

<table>
<tr><th>类　别</th><th>运　算　符</th></tr>
<tr><td>算术运算符</td><td>+　–　*　/　%</td></tr>
<tr><td>逻辑运算符</td><td>&　|　^　~　&&　||　!</td></tr>
<tr><td>字符串连接运算符</td><td>+</td></tr>
<tr><td>递增和递减运算符</td><td>++　– –</td></tr>
<tr><td>移位运算符</td><td><<　>></td></tr>
<tr><td>比较运算符</td><td>==　!=　<>　<=>=</td></tr>
<tr><td>赋值运算符</td><td>=　+=　- =　*=　/=　%=　&=　|=　^=　<<=　>>=</td></tr>
<tr><td>成员访问运算符(用于对象和结构)</td><td>.</td></tr>
<tr><td>索引运算符(用于数组和索引器)</td><td>[]</td></tr>
<tr><td>类型转换运算符</td><td>()</td></tr>
<tr><td>条件运算符(三元运算符)</td><td>?:</td></tr>
<tr><td>委托连接和删除运算符(见第 8 章)</td><td>+　–</td></tr>
<tr><td>对象创建运算符</td><td>new</td></tr>
<tr><td>类型信息运算符</td><td>sizeof　is　typeof　as</td></tr>
<tr><td>溢出异常控制运算符</td><td>checked　unchecked</td></tr>
<tr><td>间接寻址运算符</td><td>[]</td></tr>
<tr><td>名称空间别名限定符(见第 2 章)</td><td>::</td></tr>
<tr><td>空合并运算符</td><td>??</td></tr>
<tr><td>空值条件运算符</td><td>?. ?[]</td></tr>
<tr><td>标识符的名称运算符</td><td>nameof()</td></tr>
</table>

**注意：**

有 4 个运算符(sizeof、*、->和&)只能用于不安全的代码(这些代码忽略了 C#的类型安全性检查)，这些不安全的代码见第 5 章的讨论。

使用 C#运算符的一个最大缺点是，与 C 风格的语言一样，对于赋值(=)和比较(==)运算，C#使用不同的运算符。例如，下述语句表示“使 x 等于 3”：

```
x = 3;
```

如果要比较 x 和另一个值，就需要使用两个等号(==)：

```
if (x = 3) // compiler error
{
}
```

幸运的是，C#非常严格的类型安全规则防止出现常见的 C 错误，也就是在逻辑语句中使用赋值运算符代替比较运算符。在 C#中，下述语句会产生一个编译器错误：

```
if (x = 3) // compiler error
{
}
```

习惯使用与字符(&)来连接字符串的 Visual Basic 程序员必须改变这个习惯。在 C#中，使用加号(+)连接字符串，而&符号表示两个不同整数值的按位 AND 运算。|符号则在两个整数之间执行按位 OR 运算。Visual Basic 程序员可能还没有使用过取模(%)运算符，它返回除运算的余数，例如，如果 x 等于 7，则 x % 5 会返回 2。

在 C#中很少会用到指针，因此也很少用到间接寻址运算符(->)。使用它们的唯一场合是在不安全的代码块中，因为只有在此 C#才允许使用指针。指针和不安全的代码见第 17 章。

### 6.2.1　运算符的简化操作

表 6-2 列出了 C#中的全部简化赋值运算符。

表 6-2

| 简化运算符 | 等　价　于 |
|---|---|
| x++,++x | x = x + 1 |
| x--,--x | x = x–1 |
| x+= y | x = x + y |
| x - = y | x = x – y |
| x *= y | x = x * y |
| x /= y | x = x / y |
| x %= y | x = x % y |
| x >>= y | x = x >> y |
| x <<= y | x = x << y |
| x &= y | x = x & y |
| x \|= y | x = x \| y |

为什么用两个例子来分别说明++递增和--递减运算符？把运算符放在表达式的前面称为前置，把运算符放在表达式的后面称为后置。要点是注意它们的行为方式有所不同。

递增或递减运算符可以作用于整个表达式，也可以作用于表达式的内部。当 x++和++x 单独占一行时，它们的作用是相同的，对应于语句 x = x + 1。但当它们用于较长的表达式内部时，把运算符放在前面(++x)会在计算表达式之前递增 x；换言之，递增了 x 后，在表达式中使用新值进行计算。而把运算符放在后面(x++)会在计算表达式之后递增 x——使用 x 的原始值计算表达式。下面的例子使用++增量运算符说明了它们的区别(代码文件 OperatorsSample/Program.cs)：

```
int x = 5;
if (++x == 6) // true - x is incremented to 6 before the evaluation
{
  Console.WriteLine("This will execute");
}
if (x++ == 7) // false - x is incremented to 7 after the evaluation
{
  Console.WriteLine("This won't");
}
```

判断第一个 if 条件得到 true，因为在计算表达式之前，x 值从 5 递增为 6。然而，第二条 if 语句中的条件为 false，因为在计算整个表达式(x == 6)后，x 值才递增为 7。

前置运算符--x 和后置运算符 x--与此类似，但它们是递减，而不是递增。

其他简化运算符，如+=和-=，需要两个操作数，通过对第一个操作数执行算术、逻辑运算，从而改变该操作数的值。例如，下面两行代码是等价的：

```
x += 5;
x = x + 5;
```

下面介绍在 C#代码中频繁使用的基本运算符和类型强制转换运算符。

### 1. 条件运算符

条件运算符(?:)也称为三元运算符，是 if...else 结构的简化形式。其名称的出处是它带有 3 个操作数。它首先判断一个条件，如果条件为真，就返回一个值；如果条件为假，则返回另一个值。其语法如下：

```
condition ? true_value: false_value
```

其中 condition 是要判断的布尔表达式，true_value 是 condition 为真时返回的值，false_value 是 condition 为假时返回的值。

恰当地使用三元运算符，可以使程序非常简洁。它特别适合于给调用的函数提供两个参数中的一个。使用它可以把布尔值快速转换为字符串值 true 或 false。它也很适合于显示正确的单数形式或复数形式(代码文件 OperatorsSample/Program.cs)：

```
int x = 1;
string s = x + " ";
s += (x == 1 ? "man": "men");
Console.WriteLine(s);
```

如果 x 等于 1，这段代码就显示 1 man；如果 x 等于其他数，就显示其正确的复数形式。但要注意，如果结果需要本地化为不同的语言，就必须编写更复杂的例程，以考虑到不同语言的不同语法规则。

### 2. checked 和 unchecked 运算符

考虑下面的代码：

```
byte b = byte.MaxValue;
b++;
Console.WriteLine(b);
```

byte 数据类型只能包含 0~255 的数，给 byte.MaxValue 分配一个字节，得到 255。对于 255，字节中所有可用的 8 个位都得到设置：11111111。所以递增这个值会导致溢出，得到 0。

CLR 如何处理这个溢出取决于许多因素，包括编译器选项；所以只要有未预料到的溢出风险，就需要用某种方式确保得到我们希望的结果。

为此，C#提供了 checked 和 unchecked 运算符。如果把一个代码块标记为 checked，CLR 就会执行溢出检查，如果发生溢出，就抛出 OverflowException 异常。下面修改上述代码，使之包含 checked 运算符(代码文件 OperatorsSample/Program.cs)：

```
byte b = 255;
checked
{
  b++;
}
Console.WriteLine(b);
```

运行这段代码，就会得到一条错误信息：

```
System.OverflowException: Arithmetic operation resulted in an overflow.
```

使用 Advance Build Settings 中的 Visual Studio 项目设置 Check for Arithmetic Overflow/Underflow，可以对所有未标记的代码进行溢出检查。也可以直接在项目文件中改变它：

```
<PropertyGroup>
  <OutputType>Exe</OutputType>
  <TargetFramework>netcoreapp2.0</TargetFramework>
  <CheckForOverflowUnderflow>true</CheckForOverflowUnderflow>
</PropertyGroup>
```

如果要禁止溢出检查，则可以把代码标记为 unchecked：

```
byte b = 255;
unchecked
{
  b++;
}
Console.WriteLine(b);
```

在本例中不会抛出异常，但会丢失数据——因为 byte 数据类型不能包含 256，溢出的位会被丢弃，所以 b 变量得到的值是 0。

注意，unchecked 是默认行为。只有在需要把几行未检查的代码放在一个显式标记为 checked 的大代码块中时，才需要显式地使用 unchecked 关键字。

**注意：**

默认不检查上溢出和下溢出，因为执行检查会影响性能。使用 checked 作为默认设置时，每一个算术运算的结果都需要验证其值是否越界。算术运算也可以用于使用 i++的 for 循环中。为了避免这种性能影响，最好一直不使用默认设置(*Check for Arithmetic Overflow/ Underflow*)，在需要时使用 checked 运算符。

### 3. is 运算符

is 运算符可以检查对象是否与特定的类型兼容。短语“兼容”表示对象或者是该类型，或者派生自该类型。例如，要检查变量是否与 object 类型兼容，可以使用下面的代码(代码文件 OperatorsSample/Program.cs)：

```
int i = 10;
if (i is object)
{
  Console.WriteLine("i is an object");
}
```

int 和所有 C#数据类型一样，也从 object 继承而来；在本例中，表达式 i is object 将为 true，并显示相应的消息。

C# 7 扩展了具有类型匹配的 is 运算符。可以检查常量、类型和 var。下面的代码片段显示了常量检查的示例，它检查常量 42 和常量 null：

```
int i = 42;
if (i is 42)
{
  Console.WriteLine("i has the value 42");
}

object o = null;
if (o is null)
{
  Console.WriteLine("o is null");
}
```

使用具有类型匹配的 is 运算符，可以在类型的右边声明变量。如果 is 运算符返回 true，则该变量通过对该类型的对象的引用来填充。然后，可以在使用 is 运算符的 if 语句范围内使用该变量：

```
public static void AMethodUsingPatternMatching(object o)
{
  if (o is Person p)
  {
    Console.WriteLine($"o is a Person with firstname {p.FirstName}");
  }
}
//...
AMethodUsingPatternMatching (new Person("Katharina", "Nagel"));
```

### 4. as 运算符

as 运算符用于执行引用类型的显式类型转换。如果要转换的类型与指定的类型兼容，转换就会成功进行；如果类型不兼容，as 运算符就会返回 null 值。如下面的代码所示，如果 object 引用实际上不引用 string 实例，把 object 引用转换为 string 就会返回 null(代码文件 OperatorsSample/Program.cs)：

```
object o1 = "Some String";
object o2 = 5;
string s1 = o1 as string; // s1 = "Some String"
string s2 = o2 as string; // s2 = null
```

as 运算符允许在一步中进行安全的类型转换，不需要先使用 is 运算符测试类型，再执行转换。

**注意：**
is 和 as 运算符也用于继承，参见第 4 章。模式匹配和 is 运算符的更多内容参见第 13 章。

### 5. sizeof 运算符

使用 sizeof 运算符可以确定栈中值类型需要的长度(单位是字节) (代码文件 OperatorsSample/Program.cs)：

```
Console.WriteLine(sizeof(int));
```

其结果是显示数字 4，因为 int 有 4 个字节长。

如果结构体只包含值类型，也可以使用 sizeof 运算符和结构—— 如下所示的 Point 类(代码文件 OperatorsSample/Point.cs)：

```
public struct Point
{
  public Point(int x, int y)
  {
    X = x;
    Y = y;
  }
  public int X { get; }
  public int Y { get; }
}
```

**注意：**
类不能使用 sizeof 运算符。

如果对复杂类型(而非基本类型)使用 sizeof 运算符，就需要把代码放在 unsafe 块中，如下所示(代码文件 OperatorsSample/Program.cs)：

```
unsafe
{
  Console.WriteLine(sizeof(Point));
}
```

**注意：**
默认情况下不允许使用不安全的代码，需要在 csproj 项目文件中指定 AllowUnsafeBlocks。第 17 章将详细论述不安全的代码。

### 6. typeof 运算符

typeof 运算符返回一个表示特定类型的 System.Type 对象。例如，typeof(string)返回表示 System.String 类型的 Type 对象。在使用反射技术动态地查找对象的相关信息时，这个运算符很有用。第 16 章将介绍反射。

### 7. nameof 运算符

nameof 是新的 C# 6 运算符。该运算符接受一个符号、属性或方法，并返回其名称。

这个运算符如何使用？一个例子是需要一个变量的名称时，如检查参数是否为 null：

```
public void Method(object o)
{
  if (o == null) throw new ArgumentNullException(nameof(o));
```

当然，这类似于传递一个字符串而不是使用 nameof 运算符来抛出异常。然而，如果名称拼错，传递字符串并不会显示一个编译器错误。另外，改变参数的名称时，就很容易忘记更改传递到 ArgumentNullException 构造函数的字符串。

```
if (o == null) throw new ArgumentNullException("o");
```

对变量的名称使用 nameof 运算符只是一个用例。还可以使用它得到属性的名称，例如，在属性 set 访问器中触发改变事件(使用 INotifyPropertyChanged 接口)，并传递属性的名称。

```
public string FirstName
```

```
{
  get => _firstName;
  set
  {
    _firstName = value;
    OnPropertyChanged(nameof(FirstName));
  }
}
```

nameof 运算符也可以用来得到方法的名称。如果方法是重载的，它同样适用，因为所有的重载版本都得到相同的值：方法的名称。

```
public void Method()
{
  Log($"{nameof(Method)} called");
```

#### 8. index 运算符

在第 7 章“数组”中将使用索引运算符(括号)访问数组。这里传递数值 2，使用索引运算符访问数组 arr1 的第三个元素：

```
int[] arr1 = {1, 2, 3, 4};
int x = arr1[2]; // x == 3
```

类似于访问数组元素，索引运算符用集合类实现(参见第 10 章)。

索引运算符不需要把整数放在括号内，并且可以用任何类型定义。下面的代码片段创建了一个泛型字典，其键是一个字符串，值是一个整数。在字典中，键可以与索引器一起使用。在下面的示例中，字符串 first 传递给索引运算符，以设置字典里的这个元素，然后把相同的字符串传递给索引器来检索此元素：

```
var dict = new Dictionary<string, int>();
dict["first"] = 1;
int x = dict["first"];
```

> **注意：**
> 本章后面的 6.6 节“实现自定义的索引运算符”将介绍如何在自己的类中创建索引运算符。

#### 9. 可空类型和运算符

值类型和引用类型的一个重要区别是，引用类型可以为空。值类型(如 int)不能为空。把 C#类型映射到数据库类型时，这是一个特殊的问题。数据库中的数值可以为空。在早期的 C#版本中，一个解决方案是使用引用类型来映射可空的数据库数值。然而，这种方法会影响性能，因为垃圾收集器需要处理引用类型。现在可以使用可空的 int 来替代正常的 int。其开销只是使用一个额外的布尔值来检查或设置空值。可空类型仍然是值类型。

在下面的代码片段中，变量 i1 是一个 int，并给它分配 1。i2 是一个可空的 int，给它分配 i1。可空性使用?和类型来定义。给 int？分配整数值的方式类似于 i1 的分配。变量 i3 表明，也可以给可空类型分配 null(代码文件 NullableTypesSample/Program.cs)。

```
int i1 = 1;
int? i2 = 2;
int? i3 = null;
```

每个结构都可以定义为可空类型，如下面的 long?和 DateTime?所示：

```
long? l1 = null;
DateTime? d1 = null;
```

如果在程序中使用可空类型，就必须考虑 null 值在与各种运算符一起使用时的影响。通常可空类型与一元或二元运算符一起使用时，如果其中一个操作数或两个操作数都是 null，其结果就是 null。例如：

```
int? a = null;
int? b = a + 4; // b = null
int? c = a * 5; // c = null
```

但是在比较可空类型时，只要有一个操作数是 null，比较的结果就是 false。即不能因为一个条件是 false，就认为该条件的对立面是 true，这种情况在使用非可空类型的程序中很常见。例如，在下面的例子中，如果 a 是空，则

无论 b 的值是+5 还是−5，总是会调用 else 子句：

```
int? a = null;
int? b = -5;
if (a >= b) // if a or b is null, this condition is false
{
  Console.WriteLine("a >= b");
}
else
{
  Console.WriteLine("a < b");
}
```

**注意：**
null 值的可能性表示，不能随意合并表达式中的可空类型和非可空类型，详见 6.3.1 节“类型转换”的内容。

**注意：**
使用 C#关键字?和类型声明时，例如 int ?，编译器会解析它，以使用泛型类型 Nullable<int>。C#编译器把速记符号转换为泛型类型，来减少输入量。

### 10. 空合并运算符

空合并运算符(??)提供了一种快捷方式，可以在处理可空类型和引用类型时表示 null 值的可能性。这个运算符放在两个操作数之间，第一个操作数必须是一个可空类型或引用类型；第二个操作数必须与第一个操作数的类型相同，或者可以隐式地转换为第一个操作数的类型。空合并运算符的计算如下：

- 如果第一个操作数不是 null，整个表达式就等于第一个操作数的值。
- 如果第一个操作数是 null，整个表达式就等于第二个操作数的值。

例如：

```
int? a = null;
int b;
b = a ?? 10; // b has the value 10
a = 3;
b = a ?? 10; // b has the value 3
```

如果第二个操作数不能隐式地转换为第一个操作数的类型，就生成一个编译时错误。

空合并运算符不仅对可空类型很重要，对引用类型也很重要。在下面的代码片段中，属性 Val 只有在不为空时才返回_val 变量的值。如果它为空，就创建 MyClass 的一个新实例，分配给_val 变量，最后从属性中返回。只有在变量_val 为空时，才执行 get 访问器中表达式的第二部分。

```
private MyClass _val;
public MyClass Val
{
  get => _val ?? (_val = new MyClass());
}
```

### 11. 空值条件运算符

C#中减少大量代码行的一个功能是空值条件运算符。生产环境中的大量代码行都会验证空值条件。访问作为方法参数传递的成员变量之前，需要检查它，以确定该变量的值是否为 null，否则会抛出一个 NullReferenceException 异常。.NET 设计准则指定，代码不应该抛出这些类型的异常，应该检查空值条件。然而，很容易忘记这样的检查。下面的这个代码片段验证传递的参数 p 是否非空。如果它为空，方法就只是返回，而不会继续执行：

```
public void ShowPerson(Person p)
{
  if (p == null) return;
  string firstName = p.FirstName;
  //...
}
```

使用空值条件运算符访问 FirstName 属性(p?.FirstName)，当 p 为空时，就只返回 null，而不继续执行表达式的右侧(代码文件 OperatorsSample/Program.cs)。

```
public void ShowPerson(Person p)
{
  string firstName = p?.FirstName;
  //...
}
```

使用空值条件运算符访问 int 类型的属性时，不能把结果直接分配给 int 类型，因为结果可以为空。解决这个问题的一种选择是把结果分配给可空的 int：

```
nt? age = p?.Age;
```

当然，要解决这个问题，也可以使用空合并运算符，定义另一个结果(例如 0)，以防止左边的结果为空：

```
int age1 = p?.Age ?? 0;
```

也可以结合多个空值条件运算符。下面访问 Person 对象的 Address 属性，这个属性又定义了 City 属性。Person 对象需要进行 null 检查，如果它不为空，Address 属性的结果也不为空：

```
Person p = GetPerson();
string city = null;
if (p != null && p.HomeAddress != null)
{
  city = p.HomeAddress.City;
}
```

使用空值条件运算符时，代码会更简单：

```
string city = p?.HomeAddress?.City;
```

还可以把空值条件运算符用于数组。在下面的代码片段中，使用索引运算符访问值为 null 的数组变量元素时，会抛出 NullReferenceException 异常：

```
int[] arr = null;
int x1 = arr[0];
```

当然，可以进行传统的 null 检查，以避免这个异常条件。更简单的版本是使用?[0]访问数组中的第一个元素。如果结果是 null，空合并运算符就返回 x1 变量的值：

```
int x1 = arr?[0] ?? 0;
```

## 6.2.2　运算符的优先级和关联性

表 6-3 显示了 C#运算符的优先级，其中顶部的运算符有最高的优先级(即在包含多个运算符的表达式中，最先计算该运算符)。

表 6-3

| 组 | 运　算　符 |
|---|---|
| 基本运算符 | ·?·　()　[]　?[]　x++　x-- new typeof sizeof checked unchecked |
| 一元运算符 | +　-!　~　++x　--x 和数据类型强制转换 |
| 乘/除运算符 | * / % |
| 加/减运算符 | + - |
| 移位运算符 | <<　>> |
| 关系运算符 | <　><=　>=　is　as |
| 比较运算符 | ==　!= |
| 按位 AND 运算符 | & |
| 按位 XOR 运算符 | ^ |
| 按位 OR 运算符 | \| |

(续表)

| 组 | 运 算 符 |
|---|---|
| 条件 AND 运算符 | && |
| 条件 OR 运算符 | \|\| |
| 空合并运算符 | ?? |
| 条件运算符 | ?: |
| 赋值运算符和 lambda | = += -= *= /= %= &= \|= ^= <<= >>= >>>= => |

除了运算符优先级，对于二元运算符，需要注意运算符是从左向右还是从右向左计算。除了少数运算符，所有的二元运算符都是左关联的。例如：

```
x + y + z
```

就等于：

```
(x + y) + z
```

需要先注意运算符的优先级，再考虑其关联性。在以下表达式中，先计算 y 和 z 相乘，再把计算的结果分配给 x，因为乘法的优先级高于加法：

```
x + y * z
```

关联性的重要例外是赋值运算符，它们是右关联。下面的表达式从右向左计算：

```
x = y = z
```

因为存在右关联性，所有变量 x、y、z 的值都是 3，且该运算符是从右向左计算的。如果这个运算符是从左向右计算，就不会是这种情况：

```
int z = 3;
int y = 2;
int x = 1;
x = y = z;
```

一个重要的、可能误导的右关联运算符是条件运算符。表达式

```
a ? b: c ? d: e
```

等于：

```
a = b: (c ? d: e)
```

这是因为该运算符是右关联的。

**注意：**

*在复杂的表达式中，应避免利用运算符优先级来生成正确的结果。使用圆括号指定运算符的执行顺序，可以使代码更整洁，避免出现潜在的冲突。*

## 6.3 使用二进制运算符

在学习编程时，使用二进制值一直是一个需要理解的重要概念，因为计算机使用 0 和 1。现在，许多人可能已经错过了它的学习，因为他们是使用 Blocks、Scratch，甚至可能是使用 JavaScript 开始学习编程的。如果用户已经很了解 0 和 1，本节仍然可以帮助复习。

在 C# 7 中，由于使用数字分隔符和二进制字面量，因此二进制值的处理比以前更容易。第 2 章讨论了这两个特性。二进制运算符从 C#的第一个版本就开始有了，本节将介绍它们。

首先，从使用二进制运算符的简单计算开始。方法 SimpleCalculations 首先使用二进制值(二进制字面量和数字分隔符)声明并初始化变量 binary1 和 binary2。使用&运算符，两个值用二进制 ADD 运算符合并起来，并写入变量 binaryAnd。然后，使用运算符 | 创建 binaryOr 变量，使用运算符 ^ 创建 binaryXOR 变量，使用运算

符 ~ 创建 reverse1 变量(代码文件 BinaryCalculations/Program.cs)：

```
static void SimpleCalculations()
{
  Console.WriteLine(nameof(SimpleCalculations));
  uint binary1 = 0b1111_0000_1100_0011_1110_0001_0001_1000;
  uint binary2 = 0b0000_1111_1100_0011_0101_1010_1110_0111;
  uint binaryAnd = binary1 & binary2;
  DisplayBits("AND", binaryAnd, binary1, binary2);
  uint binaryOR = binary1 | binary2;
  DisplayBits("OR", binaryOR, binary1, binary2);
  uint binaryXOR = binary1 ^ binary2;
  DisplayBits("XOR", binaryXOR, binary1, binary2);
  uint reverse1 = ~binary1;
  DisplayBits("NOT", reverse1, binary1);
  Console.WriteLine();
}
```

要以二进制形式显示 uint 和 int 变量，需要创建扩展方法 ToBinaryString。Convert.ToString 提供的一个重载带有两个 int 参数，其中第二个 int 值是 toBase 参数。使用这个方法，可以通过传递值 2、八进制(8)、十进制(10)和十六进制(16)来格式化输出字符串 binary。默认情况下，如果二进制值以 0 开始，这些 0 值将被忽略，而不会打印出来。PadLeft 方法填充字符串中的这些 0 值。字符串需要的字符数由 sizeof 运算符计算，并左移 4 位。如前所述，sizeof 运算符返回指定类型的字节数。要显示这些位，需要将字节数乘以 8，这相当于向左移动 3 位。另一个扩展方法是 AddSeparators，它使用 LINQ 方法在每四位数之后添加_分隔符(代码文件 BinaryCalculations/BinaryExtensions.cs)：

```
public static class BinaryExtensions
{
  public static string ToBinaryString(this uint number) =>
    Convert.ToString(number, toBase: 2).PadLeft(sizeof(uint) << 3, '0');
  public static string ToBinaryString(this int number) =>
    Convert.ToString(number, toBase: 2).PadLeft(sizeof(int) << 3, '0');

  public static string AddSeparators(this string number) =>
    string.Join('_',
      Enumerable.Range(0, number.Length / 4)
        .Select(i => number.Substring(i * 4, 4)).ToArray());
}
```

**注意：**

AddSeparators 使用 LINQ。LINQ 详见第 12 章。

方法 DisplayBits 是从前面显示的 SimpleCalculations 方法调用的，它使用 ToBinaryString 和 AddSeparators 扩展方法。在这里，将显示用于操作的操作数，以及结果(代码文件 BinaryCalculations/Program.cs)：

```
static void DisplayBits(string title, uint result, uint left,
  uint? right = null)
{
  Console.WriteLine(title);
  Console.WriteLine(left.ToBinaryString().AddSeparators());
  if (right.HasValue)
  {
    Console.WriteLine(right.Value.ToBinaryString().AddSeparators());
  }
  Console.WriteLine(result.ToBinaryString().AddSeparators());
  Console.WriteLine();
}
```

在运行应用程序时，可以看到使用二进制运算符&的以下输出。对于这个运算符，只有两个输入值都为 1 时，得到的位才是 1：

```
AND
1111_0000_1100_0011_1110_0001_0001_1000
0000_1111_1100_0011_0101_1010_1110_0111
0000_0000_1100_0011_0100_0000_0000_0000
```

应用二进制运算符|，如果设置一个输入位，则设置结果位(1)：

```
OR
1111_0000_1100_0011_1110_0001_0001_1000
```

```
0000_1111_1100_0011_0101_1010_1110_0111
1111_1111_1100_0011_1111_1011_1111_1111
```

对于^运算符，如果两个原始的位只设置了一个，而没有设置两个，就设置结果：

```
XOR
1111_0000_1100_0011_1110_0001_0001_1000
0000_1111_1100_0011_0101_1010_1110_0111
1111_1111_0000_0000_1011_1011_1111_1111
```

最后，对于运算符~，结果是对原始位的否定：

```
NOT
1111_0000_1100_0011_1110_0001_0001_1000
0000_1111_0011_1100_0001_1110_1110_0111
```

### 6.3.1　位的移动

如前面的示例所述，向左移动 3 位就是原来的数字乘以 8。向左移动 1 位就是原来的数字乘以 2。这比调用乘法运算符要快得多——假定需要乘以 2、4、8、16、32 等。

下面的代码片段在变量 s1 中设置了一个位，在 for 循环中，这个位总是移动一位(代码文件 BinaryCalculations/Program.cs)：

```
static void ShiftingBits()
{
  Console.WriteLine(nameof(ShiftingBits));
  ushort s1 = 0b01;
  for (int i = 0; i < 16; i++)
  {
    Console.WriteLine($"{s1.ToBinarString()} {s1} hex: {s1:X}");
    s1 = (ushort)(s1 << 1);
  }
  Console.WriteLine();
}
```

在程序的输出中，可以看到循环中的二进制、十进制和十六进制值：

```
0000000000000001 1 hex: 1
0000000000000010 2 hex: 2
0000000000000100 4 hex: 4
0000000000001000 8 hex: 8
0000000000010000 16 hex: 10
0000000000100000 32 hex: 20
0000000001000000 64 hex: 40
0000000010000000 128 hex: 80
0000000100000000 256 hex: 100
0000001000000000 512 hex: 200
0000010000000000 1024 hex: 400
0000100000000000 2048 hex: 800
0001000000000000 4096 hex: 1000
0010000000000000 8192 hex: 2000
0100000000000000 16384 hex: 4000
1000000000000000 32768 hex: 8000
```

### 6.3.2　有符号数和无符号数

使用二进制时要记住的一件重要的事情是，使用带符号的类型时，如 int、long、short，最左端的一位用来表示符号。使用 int 类型时，可用的最大值是 2147483647 —— 31 位的正数或 0x7FFF FFFF。对于 uint，可用的最大值是 4294967295 或 0xFFFF FFFF。这表示 32 位的正数。对于 int，数字范围的另一半用于负数。

为了理解负数是如何表示的，下面的代码片段使用 int.MaxValue 将 maxNumber 变量初始化为最大的 31 位正数。然后，在 for 循环中，该变量会递增三次。在所有的结果中，都将显示二进制、十进制和十六进制值(代码文件 BinaryCalculations/Program.cs)：

```
private static void SignedNumbers()
{
  Console.WriteLine(nameof(SignedNumbers));

  void DisplayNumber(string title, int x) =>
    Console.WriteLine($"{title,-11} " +
      $"bin: {x.ToBinaryString().AddSeparators()}, " +
```

```
      $"dec: {x}, hex: {x:X}");

  int maxNumber = int.MaxValue;
  DisplayNumber("max int", maxNumber);
  for (int i = 0; i < 3; i++)
  {
    maxNumber++;
    DisplayNumber($"added {i + 1}", maxNumber);
  }
  Console.WriteLine();
  //...
}
```

在应用程序的输出中可以看到，除符号位之外的所有位都设置了，得到了最大的整数值。输出以不同的格式显示相同的值——二进制、十进制和十六进制。在第一个输出中添加 1，将导致设置符号位的 int 类型溢出，其他所有位都是 0。这是 int 类型的最大负值。在这个结果之后，又递增了两次：

```
max int    bin: 0111_1111_1111_1111_1111_1111_1111_1111, dec: 2147483647,
  hex: 7FFFFFFF
added 1    bin: 1000_0000_0000_0000_0000_0000_0000_0000, dec: -2147483648,
  hex: 80000000
added 2    bin: 1000_0000_0000_0000_0000_0000_0000_0001, dec: -2147483647,
  hex: 80000001
added 3    bin: 1000_0000_0000_0000_0000_0000_0000_0010, dec: -2147483646,
  hex: 80000002
```

在下一个代码片段中，变量 zero 初始化为 0。在 for 循环中，这个变量递减三次：

```
int zero = 0;
DisplayNumber("zero", zero);
for (int i = 0; i < 3; i++)
{
  zero--;
  DisplayNumber($"subtracted {i + 1}", zero);
}
Console.WriteLine();
```

在输出中可以看到，所有未设置的位都表示为 0。递减的结果是十进制-1，它设置了所有位，包括符号位：

```
zero         bin: 0000_0000_0000_0000_0000_0000_0000_0000, dec: 0, hex: 0
subtracted 1 bin: 1111_1111_1111_1111_1111_1111_1111_1111, dec: -1, hex: FFFFFFFF
subtracted 2 bin: 1111_1111_1111_1111_1111_1111_1111_1110, dec: -2, hex: FFFFFFFE
subtracted 3 bin: 1111_1111_1111_1111_1111_1111_1111_1101, dec: -3, hex: FFFFFFFD
```

接下来，int 从最大的负数开始，这个数字递增了三次：

```
int minNumber = int.MinValue;
DisplayNumber("min number", minNumber);
for (int i = 0; i < 3; i++)
{
  minNumber++;
  DisplayNumber($"added {i + 1}", minNumber);
}
Console.WriteLine();
```

如前所述，当溢出最大的正数时，就会显示最大的负数。在使用 int.MinValue 时，就会看到相同的数字。这个数字递增了三次：

```
min number  bin: 1000_0000_0000_0000_0000_0000_0000_0000, dec: -2147483648,
  hex: 80000000
added 1    bin: 1000_0000_0000_0000_0000_0000_0000_0001, dec: -2147483647,
  hex: 80000001
added 2    bin: 1000_0000_0000_0000_0000_0000_0000_0010, dec: -2147483646,
  hex: 80000002
added 3    bin: 1000_0000_0000_0000_0000_0000_0000_0011, dec: -2147483645,
  hex: 80000003
```

## 6.4　类型的安全性

第 1 章提到，中间语言(IL)可以对其代码强制实现强类型安全性。强类型化支持.NET 提供的许多服务，包括安全性和语言的交互性。因为 C#语言会编译为 IL，所以 C#也是强类型的。此外，这说明数据类型并不总是可无缝互换。本节将介绍基本类型之间的转换。

**注意：**

C#也支持不同引用类型之间的转换，在与其他类型相互转换时还允许定义所创建的数据类型的行为方式。本章稍后将详细讨论这两个主题。

另一方面，泛型可以避免对一些常见的情形进行类型转换，详见第 5 章。

### 6.4.1 类型转换

我们常常需要把数据从一种类型转换为另一种类型。考虑下面的代码：

```
byte value1 = 10;
byte value2 = 23;
byte total;
total = value1 + value2;
Console.WriteLine(total);
```

在试图编译这些代码行时，会得到一条错误消息：

```
Cannot implicitly convert type 'int' to 'byte'
```

问题是，把两个 byte 型数据加在一起时，应返回 int 型结果，而不是另一个 byte 数据。这是因为 byte 包含的数据只能为 8 位，所以把两个 byte 型数据加在一起，很容易得到不能存储在单个 byte 型数据中的值。如果要把结果存储在一个 byte 变量中，就必须把它转换回 byte 类型。C#支持两种转换方式：隐式转换和显式转换。

#### 1. 隐式转换

只要能保证值不会发生任何变化，类型转换就可以自动(隐式)进行。这就是前面代码失败的原因：试图从 int 转换为 byte，而可能丢失了 3 个字节的数据。编译器不允许这么做，除非我们明确告诉它这就是我们希望的结果！如果在 long 类型变量而非 byte 类型变量中存储结果，就不会有问题了：

```
byte value1 = 10;
byte value2 = 23;
long total; // this will compile fine
total = value1 + value2;
Console.WriteLine(total);
```

程序可以顺利编译，而没有任何错误，这是因为 long 类型变量包含的数据字节比 byte 类型多，所以没有丢失数据的危险。在这些情况下，编译器会很顺利地转换，我们也不需要显式地提出要求。

表 6-4 列出了 C#支持的隐式类型转换。

表 6-4

| 源 类 型 | 目 标 类 型 |
|---|---|
| sbyte | short、int、long、float、double、decimal、BigInteger |
| byte | short、ushort、int、uint、long、ulong、float、double、decimal、BigInteger |
| short | int、long、float、double、decimal、BigInteger |
| ushort | int、uint、long、ulong、float、double、decimal、BigInteger |
| int | long、float、double、decimal、BigInteger |
| uint | long、ulong、float、double、decimal、BigInteger |
| long、ulong | float、double、decimal、BigInteger |
| float | double、BigInteger |
| char | ushort、int、uint、long、ulong、float、double、decimal、BigInteger |

**注意：**

BigInteger 是包含任意大小的数字的结构体。可以从较小的类型中初始化它，传递一个数字数组来创建一个大的数字，或者解析包含大数字的字符串。这种类型实现了数学计算的方法。BigInteger 的名称空间是 System.Numeric。

注意，只能从较小的整数类型隐式地转换为较大的整数类型，而不能从较大的整数类型隐式地转换为较小的整数类型。也可以在整数和浮点数之间转换；然而，其规则略有不同。尽管可以在相同大小的类型之间转换，如 int/uint 转换为 float，long/ulong 转换为 double，也可以从 long/ulong 转换回 float。这样做可能会丢失 4 个字节的数据，但这仅表示得到的 float 值比使用 double 得到的值精度低；编译器认为这是一种可以接受的错误，因为值的数量级不会受到影响。还可以将无符号的变量分配给有符号的变量，只要无符号变量值的大小在有符号变量的范围之内即可。

在隐式地转换值类型时，对于可空类型需要考虑其他因素：

- 可空类型隐式地转换为其他可空类型，应遵循表 6-4 中非可空类型的转换规则。即 int?隐式地转换为 long?、float?、double?和 decimal?。
- 非可空类型隐式地转换为可空类型也遵循表 6-4 中的转换规则，即 int 隐式地转换为 long?、float?、double?和 decimal?。
- 可空类型不能隐式地转换为非可空类型，此时必须进行显式转换，如下一节所述。这是因为可空类型的值可以是 null，但非可空类型不能表示这个值。

### 2. 显式转换

有许多场合不能隐式地转换类型，否则编译器会报告错误。下面是不能进行隐式转换的一些场合：

- int 转换为 short——会丢失数据。
- int 转换为 uint——会丢失数据。
- uint 转换为 int——会丢失数据。
- float 转换为 int——会丢失小数点后面的所有数据。
- 任何数字类型转换为 char——会丢失数据。
- decimal 转换为任何数字类型——因为 decimal 类型的内部结构不同于整数和浮点数。
- int?转换为 int——可空类型的值可以是 null。

但是，可以使用类型强制转换(cast)显式地执行这些转换。在把一种类型强制转换为另一种类型时，有意地迫使编译器进行转换。类型强制转换的一般语法如下：

```
long val = 30000;
int i = (int)val; // A valid cast. The maximum int is 2147483647
```

这表示，把强制转换的目标类型名放在要转换值之前的圆括号中。对于熟悉 C 的程序员，这是类型强制转换的典型语法。对于熟悉 C++类型强制转换关键字(如 static_cast)的程序员，这些关键字在 C#中不存在，必须使用 C 风格的旧语法。

这种类型强制转换是一种比较危险的操作，即使在从 long 转换为 int 这样简单的类型强制转换过程中，如果原来 long 的值比 int 的最大值还大，就会出现问题：

```
long val = 3000000000;
int i = (int)val; // An invalid cast. The maximum int is 2147483647
```

在本例中，不会报告错误，但也得不到期望的结果。如果运行上面的代码，并将输出结果存储在 i 中，则其值为：

```
-1294967296
```

最好假定显式类型强制转换不会给出希望的结果。如前所述，C#提供了一个 checked 运算符，使用它可以测试操作是否会导致算术溢出。使用 checked 运算符可以检查类型强制转换是否安全，如果不安全，就要迫使运行库抛出一个溢出异常：

```
long val = 3000000000;
int i = checked((int)val);
```

记住，所有的显式类型强制转换都可能不安全，在应用程序中应包含代码来处理可能失败的类型强制转换。第 14 章将使用 try 和 catch 语句引入结构化异常处理。

使用类型强制转换可以把大多数基本数据类型从一种类型转换为另一种类型。例如，下面的代码给 price

加上 0.5，再把结果强制转换为 int：

```
double price = 25.30;
int approximatePrice = (int)(price + 0.5);
```

这会把价格四舍五入为最接近的美元数。但在这个转换过程中，小数点后面的所有数据都会丢失。因此，如果要使用这个修改过的价格进行更多的计算，最好不要使用这种转换；如果要输出全部计算或部分计算的近似值，且不希望由于小数点后面的多位数据而麻烦用户，这种转换就很合适。

下面的例子说明了把无符号整数转换为 char 时会发生的情况：

```
ushort c = 43;
char symbol = (char)c;
Console.WriteLine(symbol);
```

输出结果是 ASCII 码为 43 的字符，即+符号。可以尝试数字类型(包括 char)之间的任何转换，这种转换是可行的，例如，把 decimal 转换为 char，或把 char 转换为 decimal。

值类型之间的转换并不仅限于孤立的变量。还可以把类型为 double 的数组元素转换为类型为 int 的结构成员变量：

```
struct ItemDetails
{
  public string Description;
  public int ApproxPrice;
}
//...
double[] Prices = { 25.30, 26.20, 27.40, 30.00 };
ItemDetails id;
id.Description = "Hello there.";
id.ApproxPrice = (int)(Prices[0] + 0.5);
```

要把一个可空类型转换为非可空类型，或转换为另一个可空类型，并且其中可能会丢失数据，就必须使用显式的类型强制转换。甚至在底层基本类型相同的元素之间进行转换时，也要使用显式的类型强制转换。例如，int?转换为 int，或 float?转换为 float。这是因为可空类型的值可以是 null，而非可空类型不能表示这个值。只要可以在两种等价的非可空类型之间进行显式的类型强制转换，对应可空类型之间显式的类型强制转换就可以进行。但如果从可空类型强制转换为非可空类型，且变量的值是 null，就会抛出 InvalidOperationException 异常。例如：

```
int? a = null;
int b = (int)a; // Will throw exception
```

谨慎地使用显式的类型强制转换，就可以把简单值类型的任何实例转换为几乎任何其他类型。但在进行显式的类型转换时有一些限制，就值类型来说，只能在数字、char 类型和 enum 类型之间转换。不能直接把布尔型强制转换为其他类型，也不能把其他类型转换为布尔型。

如果需要在数字和字符串之间转换，就可以使用.NET 类库中提供的一些方法。Object 类实现了一个 ToString()方法，该方法在所有的.NET 预定义类型中都进行了重写，并返回对象的字符串表示：

```
int i = 10;
string s = i.ToString();
```

同样，如果需要分析一个字符串，以检索一个数字或布尔值，就可以使用所有预定义值类型都支持的 Parse()方法：

```
string s = "100";
int i = int.Parse(s);
Console.WriteLine(i + 50); // Add 50 to prove it is really an int
```

注意，如果不能转换字符串(例如，要把字符串 Hello 转换为一个整数)，Parse()方法就会通过抛出一个异常，注册一个错误。第 14 章将介绍异常。

### 6.4.2 装箱和拆箱

第 2 章介绍了所有类型，包括简单的预定义类型(如 int 和 char)和复杂类型(如从 object 类型中派生的类和结构)。这意味着可以像处理对象那样处理字面值：

```
string s = 10.ToString();
```

但是，C#数据类型可以分为在栈上分配内存的值类型和在托管堆上分配内存的引用类型。如果 int 不过是栈上一个 4 字节的值，该如何在它上面调用方法？

C#的实现方式是通过一个有点魔术性的方式，即装箱(boxing)。装箱和拆箱(unboxing)可以把值类型转换为引用类型，并把引用类型转换回值类型。这些操作包含在 6.7 节中，因为它们是基本的操作，即把值强制转换为 object 类型。装箱用于描述把一个值类型转换为引用类型。运行库会为堆上的对象创建一个临时的引用类型“箱子”。

该转换可以隐式地进行，如上面的例子所述。还可以显式地进行转换：

```
int myIntNumber = 20;
object myObject = myIntNumber;
```

拆箱用于描述相反的过程，其中以前装箱的值类型强制转换回值类型。这里使用术语“强制转换”，是因为这种转换是显式进行的。其语法类似于前面的显式类型转换：

```
int myIntNumber = 20;
object myObject = myIntNumber; // Box the int
int mySecondNumber = (int)myObject; // Unbox it back into an int
```

只能对以前装箱的变量进行拆箱。当 myObject 不是装箱的 int 类型时，如果执行最后一行代码，就会在运行期间抛出一个运行时异常。

这里有一个警告：在拆箱时必须非常小心，确保得到的值变量有足够的空间存储拆箱的值中的所有字节。例如，C#的 int 类型只有 32 位，所以把 long 值(64 位)拆箱为 int 时，会导致抛出一个 InvalidCastException 异常：

```
long myLongNumber = 333333423;
object myObject = (object)myLongNumber;
int myIntNumber = (int)myObject;
```

# 6.5　比较对象的相等性

在讨论了运算符并简要介绍了相等运算符后，就应考虑在处理类和结构的实例时，“相等”意味着什么。理解对象相等的机制对逻辑表达式的编程非常重要，另外对实现运算符重载和类型强制转换也非常重要，本章后面将讨论运算符重载。

对象相等的机制有所不同，这取决于比较的是引用类型(类的实例)还是值类型(基本数据类型、结构或枚举的实例)。下面分别介绍引用类型和值类型的相等性。

## 6.5.1　比较引用类型的相等性

System.Object 定义了 3 个不同的方法来比较对象的相等性：ReferenceEquals()和两个版本的 Equals()：一个是静态的方法，一个是可以重写的虚拟实例方法。还可以实现接口 IEquality<T>，它提供了一个具有泛型类型参数而不是对象的 Equals 方法。再加上比较运算符(==)，实际上有 4 种比较相等性的方法。这些方法有一些细微的区别，下面就介绍它们。

### 1. ReferenceEquals()方法

ReferenceEquals()是一个静态方法，其测试两个引用是否指向类的同一个实例，特别是两个引用是否包含内存中的相同地址。作为静态方法，它不能重写，所以 System.Object 的实现代码保持不变。如果提供的两个引用指向同一个对象实例，则 ReferenceEquals()总是返回 true；否则就返回 false。但是，它认为 null 等于 null(代码文件 EqualsSample/Program.cs)：

```
static void ReferenceEqualsSample()
{
  SomeClass x = new SomeClass(), y = new SomeClass(), z = x;

  bool b1 = object.ReferenceEquals(null, null); // returns true
  bool b2 = object.ReferenceEquals(null, x);    // returns false
```

```
    bool b3 = object.ReferenceEquals(x, y);        // returns false because x and y
                                                   // references different objects
    bool b4 = object.ReferenceEquals(x, z);        // returns true because x and z
                                                   // references the same object
    //...
  }
```

### 2. Equals()虚方法

Equals()虚版本的 System.Object 实现代码也可以比较引用。但因为这是虚方法，所以可以在自己的类中重写它，从而按值来比较对象。特别是如果希望类的实例用作字典中的键，就需要重写这个方法，以比较相关值。否则，根据重写 Object.GetHashCode()的方式，包含对象的字典类要么不工作，要么工作的效率非常低。在重写 Equals()方法时要注意，重写的代码不应抛出异常。同理，这是因为如果抛出异常，字典类就会出问题，一些在内部调用这个方法的.NET 基类也可能出问题。

### 3. 静态的 Equals()方法

Equals()的静态版本与其虚实例版本的作用相同，其区别是静态版本带有两个参数，并对它们进行相等性比较。这个方法可以处理两个对象中有一个是 null 的情况；因此，如果一个对象可能是 null，这个方法就可以抛出异常，提供额外的保护。静态重载版本首先要检查传递给它的引用是否为 null。如果它们都是 null，就返回 true(因为 null 与 null 相等)。如果只有一个引用是 null，它就返回 false。如果两个引用实际上引用了某个对象，它就调用 Equals()的虚实例版本。这表示在重写 Equals()的实例版本时，其效果相当于也重写了静态版本。

### 4. 比较运算符(==)

最好将比较运算符看作严格的值比较和严格的引用比较之间的中间选项。在大多数情况下，下面的代码表示正在比较引用：

```
bool b = (x == y); // x, y object references
```

但是，如果把一些类看作值，其含义就会比较直观，这是可以接受的方法。在这些情况下，最好重写比较运算符，以执行值的比较。后面将讨论运算符的重载，但一个明显例子是 System.String 类，Microsoft 重写了这个运算符，以比较字符串的内容，而不是比较它们的引用。

## 6.5.2　比较值类型的相等性

在比较值类型的相等性时，采用与引用类型相同的规则：ReferenceEquals()用于比较引用，Equals()用于比较值，比较运算符可以看作一个中间项。但最大的区别是值类型需要装箱，才能把它们转换为引用，进而才能对它们执行方法。另外，Microsoft 已经在 System.ValueType 类中重载了实例方法 Equals()，以便对值类型进行合适的相等性测试。如果调用 sA.Equals(sB)，其中 sA 和 sB 是某个结构的实例，则根据 sA 和 sB 是否在其所有的字段中包含相同的值而返回 true 或 false。另一方面，在默认情况下，不能对自己的结构重载==运算符。在表达式中使用(sA == sB)会导致一个编译错误，除非在代码中为当前的结构提供了==的重载版本。

另外，ReferenceEquals()在应用于值类型时总是返回 false，因为为了调用这个方法，值类型需要装箱到对象中。即使编写下面的代码：

```
bool b = ReferenceEquals(v,v); // v is a variable of some value type
```

也会返回 false，因为在转换每个参数时，v 都会被单独装箱，这意味着会得到不同的引用。出于上述原因，调用 ReferenceEquals()来比较值类型实际上没有什么意义，所以不能调用它。

尽管 System.ValueType 提供的 Equals()默认重写版本肯定足以应付绝大多数自定义的结构，但仍可以针对自己的结构再次重写它，以提高性能。另外，如果值类型包含作为字段的引用类型，就需要重写 Equals()，以便为这些字段提供合适的语义，因为 Equals()的默认重写版本仅比较它们的地址。

# 6.6　运算符重载

本节将介绍为类或结构定义的另一种类型的成员：运算符重载。C++开发人员应很熟悉运算符重载。但是，因为这对于 Java 和 Visual Basic 开发人员来说是全新的概念，所以这里要解释一下。C++开发人员可以直接跳到主要的运算符重载示例上。

运算符重载的关键是在对象上不能总是只调用方法或属性，有时还需要做一些其他工作，例如对数值进行相加、相乘或逻辑操作(如比较对象)等。假定已经定义了一个表示数学矩阵的类。在数学领域中，矩阵可以相加和相乘，就像数字一样。所以可以编写下面的代码：

```
Matrix a, b, c;
// assume a, b and c have been initialized
Matrix d = c * (a + b);
```

通过重载运算符，就可以告诉编译器，+和*对 Matrix 对象执行什么操作，以便编写类似于上面的代码。如果用不支持运算符重载的语言编写代码，就必须定义一个方法，以执行这些操作。结果肯定不太直观，可能如下所示：

```
Matrix d = c.Multiply(a.Add(b));
```

学习到现在可以知道，像+和*这样的运算符只能用于预定义的数据类型，原因很简单：编译器知道所有常见的运算符对于这些数据类型的含义。例如，它知道如何把两个 long 数据加起来，或者如何对两个 double 数据执行相除操作，并且可以生成合适的中间语言代码。但在定义自己的类或结构时，必须告诉编译器：什么方法可以调用，每个实例存储了什么字段等所有信息。同样，如果要对自定义的类使用运算符，就必须告诉编译器相关的运算符在这个类的上下文中的含义。此时就要定义运算符的重载。

要强调的另一个问题是重载不仅仅限于算术运算符。还需要考虑比较运算符 ==、<、>、!=、>=和<=。例如，考虑语句 if(a==b)。对于类，这条语句在默认状态下会比较引用 a 和 b。检测这两个引用是否指向内存中的同一个地址，而不是检测两个实例实际上是否包含相同的数据。对于 string 类，这种行为就会重写，于是比较字符串实际上就是比较每个字符串的内容。可以对自己的类进行这样的操作。对于结构，==运算符在默认状态下不做任何工作。试图比较两个结构，看看它们是否相等，就会产生一个编译错误，除非显式地重载了==，告诉编译器如何进行比较。

在许多情况下，重载运算符用于生成可读性更高、更直观的代码，包括：

- 在数学领域中，几乎包括所有的数学对象：坐标、矢量、矩阵、张量(tensor)和函数等。如果编写一个程序执行某些数学或物理建模，就几乎肯定会用类表示这些对象。
- 图形程序在计算屏幕上的位置时，也使用与数学或坐标相关的对象。
- 表示大量金钱的类(例如，在财务程序中)。
- 字处理或文本分析程序也有表示语句、子句等方面的类，可以使用运算符合并语句(这是字符串连接的一种比较复杂的版本)。

但是，也有许多类型与运算符重载并不相关。不恰当地使用运算符重载，会使使用类型的代码更难理解。例如，把两个 DateTime 对象相乘，在概念上没有任何意义。

## 6.6.1　运算符的工作方式

为了理解运算符是如何重载的，考虑一下在编译器遇到运算符时会发生什么情况就很有用。用加法运算符(+)作为例子，假定编译器处理下面的代码：

```
int myInteger = 3;
uint myUnsignedInt = 2;
double myDouble = 4.0;
long myLong = myInteger + myUnsignedInt;
double myOtherDouble = myDouble + myInteger;
```

考虑当编译器遇到下面这行代码时会发生什么情况：

```
long myLong = myInteger + myUnsignedInt;
```

编译器知道它需要把两个整数加起来，并把结果赋予一个 long 型变量。调用一个方法把数字加在一起时，表达式 myInteger + myUnsignedInt 是一种非常直观和方便的语法。该方法接受两个参数 myInteger 和 myUnsignedInt，并返回它们的和。所以编译器完成的任务与任何方法调用一样——它会根据参数类型查找最匹配的+运算符重载，这里是带两个整数参数的+运算符重载。与一般的重载方法一样，预定义的返回类型不会因为编译器调用方法的哪个版本而影响其选择。在本例中调用的重载方法接受两个 int 参数，返回一个 int 值，这个返回值随后会转换为 long 类型。

下一行代码让编译器使用+运算符的另一个重载版本：

```
double myOtherDouble = myDouble + myInteger;
```

在这个实例中，参数是一个 double 类型的数据和一个 int 类型的数据，但+运算符没有这种复合参数的重载形式，所以编译器认为，最匹配的+运算符重载是把两个 double 数据作为其参数的版本，并隐式地把 int 强制转换为 double。把两个 double 数据加在一起与把两个整数加在一起完全不同，浮点数存储为一个尾数和一个指数。把它们加在一起要按位移动一个 double 数据的尾数，从而使两个指数有相同的值，然后把尾数加起来，移动所得到尾数的位，调整其指数，保证答案有尽可能高的精度。

现在，看看如果编译器遇到下面的代码会发生什么：

```
Vector vect1, vect2, vect3;
// initialize vect1 and vect2
vect3 = vect1 + vect2;
vect1 = vect1 * 2;
```

其中，Vector 是结构，稍后再定义它。编译器知道它需要把两个 Vector 实例加起来，即 vect1 和 vect2。它会查找+运算符的重载，该重载版本把两个 Vector 实例作为参数。

如果编译器找到这样的重载版本，它就调用该运算符的实现代码。如果找不到，它就要看看有没有可以用作最佳匹配的其他+运算符重载，例如，某个运算符重载对应的两个参数是其他数据类型，但可以隐式地转换为 Vector 实例。如果编译器找不到合适的运算符重载，就会产生一个编译错误，就像找不到其他方法调用的合适重载版本一样。

### 6.6.2 运算符重载的示例：Vector 结构

本小节将开发一个结构 Vector 来说明运算符重载，这个 Vector 结构表示一个三维数学矢量。如果数学不是你的强项，不必担心，我们会使这个例子尽可能简单。就此处而言，三维矢量只是 3 个(double)数字的集合，说明物体的移动速度。表示数字的变量是_x、_y 和_z，_x 表示物体向东移动的速度，_y 表示物体向北移动的速度，_z 表示物体向上移动的速度(高度)。把这 3 个数字组合起来，就得到总移动量。例如，如果_x=3.0、_y=3.0、_z=1.0，一般可以写作(3.0, 3.0, 1.0)，表示物体向东移动 3 个单位，向北移动 3 个单位，向上移动 1 个单位。

矢量可以与其他矢量或数字相加或相乘。在这里我们还使用术语“标量”，它是简单数字的数学用语——在 C#中就是一个 double 数据。相加的作用很明显。如果先移动(3.0, 3.0, 1.0)矢量对应的距离，再移动(2.0, –4.0, –4.0)矢量对应的距离，总移动量就是把这两个矢量加起来。矢量的相加指把每个对应的组成元素分别相加，因此得到(5.0, –1.0, –3.0)。此时，数学表达式总是写成 c=a+b，其中 a 和 b 是矢量，c 是结果矢量。这与 Vector 结构的使用方式一样。

**注意：**
这个例子将作为一个结构而不是类来开发，但这并不重要。运算符重载用于结构和类时，其工作方式是一样的。

下面是 Vector 的定义——包含只读属性、构造函数和重写的 ToString()方法，以便轻松地查看 Vector 的内容，最后是运算符重载(代码文件 OperatorOverloadingSample/Vector.cs)：

```
struct Vector
```

```
{
  public Vector(double x, double y, double z)
  {
    X = x;
    Y = y;
    Z = z;
  }

  public Vector(Vector v)
  {
    X = v.X;
    Y = v.Y;
    Z = v.Z;
  }

  public double X { get; }
  public double Y { get; }
  public double Z { get; }
  public override string ToString() => $"( {X}, {Y}, {Z} )";
}
```

这里提供了两个构造函数，通过传递每个元素的值或者提供另一个复制其值的 Vector 来指定矢量的初始值。第二个构造函数带一个 Vector 参数，通常称为复制构造函数，因为它们允许通过复制另一个实例来初始化一个类或结构实例。

下面是 Vector 结构的有趣部分——为+运算符提供支持的运算符重载：

```
public static Vector operator +(Vector left, Vector right) =>
new Vector(left.X + right.X, left.Y + right.Y, left.Z + right.Z);
```

运算符重载的声明方式与静态方法基本相同，但 operator 关键字告诉编译器，它实际上是一个自定义的运算符重载，后面是相关运算符的实际符号，在本例中就是+。返回类型是在使用这个运算符时获得的类型。在本例中，把两个矢量加起来会得到另一个矢量，所以返回类型也是 Vector。对于这个特定的+运算符重载，返回类型与包含的类一样，但并不一定是这种情况，在本示例中稍后将看到。两个参数就是要操作的对象。对于二元运算符(带两个参数)，如+和-运算符，第一个参数是运算符左边的值，第二个参数是运算符右边的值。

这个实现代码返回一个新的矢量，该矢量用 left 和 right 变量的 X、Y 和 Z 属性初始化。

C#要求所有的运算符重载都声明为 public 和 static，这表示它们与其类或结构相关联，而不是与某个特定实例相关联，所以运算符重载的代码体不能访问非静态类成员，也不能访问 this 标识符；这是可行的，因为参数提供了运算符执行其任务所需要知道的所有输入数据。

下面需要编写一些简单的代码来测试 Vector 结构(代码文件 OperatorOverloadingSample/Program.cs)：

```
static void Main()
{
  Vector vect1, vect2, vect3;
  vect1 = new Vector(3.0, 3.0, 1.0);
  vect2 = new Vector(2.0, -4.0, -4.0);
  vect3 = vect1 + vect2;
  Console.WriteLine($"vect1 = {vect1}");
  Console.WriteLine($"vect2 = {vect2}");
  Console.WriteLine($"vect3 = {vect3}");
}
```

编译并运行这些代码，结果如下：

```
vect1 = ( 3, 3, 1 )
vect2 = ( 2, -4, -4 )
vect3 = ( 5, -1, -3 )
```

矢量除了可以相加之外，还可以相乘、相减和比较它们的值。本节通过添加几个运算符重载，扩展了这个 Vector 例子。这并不是一个功能齐全的真实的 Vector 类型，但足以说明运算符重载的其他方面了。首先要重载乘法运算符，以支持标量和矢量的相乘以及矢量和矢量的相乘。

矢量乘以标量只意味着矢量的每个组成元素分别与标量相乘，例如，2*(1.0, 2.5, 2.0)返回(2.0, 5.0, 4.0)。相关的运算符重载如下所示(代码文件 OperatorOverloadingSample2/Vector.cs)：

```
public static Vector operator *(double left, Vector right) =>
new Vector(left * right.X, left * right.Y, left * right.Z);
```

但这还不够，如果 a 和 b 声明为 Vector 类型，就可以编写下面的代码：

```
b = 2 * a;
```

编译器会隐式地把整数 2 转换为 double 类型，以匹配运算符重载的签名。但不能编译下面的代码：

```
b = a * 2;
```

编译器处理运算符重载的方式与方法重载是一样的。它会查看给定运算符的所有可用重载，找到与之最匹配的重载方式。上面的语句要求第一个参数是 Vector，第二个参数是整数，或者可以隐式转换为整数的其他数据类型。我们没有提供这样一个重载。有一个运算符重载，其参数依次是一个 double 和一个 Vector，但编译器不能交换参数的顺序，所以这是不可行的。需要显式地定义一个运算符重载，其参数依次是一个 Vector 和一个 double，有两种方式可以实现这样的运算符重载。第一种方式是对矢量乘法进行分解，和处理所有运算符的方式一样，显式执行矢量相乘操作：

```
public static Vector operator *(Vector left, double right) =>
  new Vector(right * left.X, right * left.Y, right * left.Z);
```

前面已经编写了实现基本相乘操作的代码，最好重用该代码：

```
public static Vector operator *(Vector left, double right) => right * left;
```

这段代码会有效地告诉编译器，如果有 Vector 和 double 数据的相乘操作，编译器就颠倒参数的顺序，调用另一个运算符重载。本章的示例代码使用第二个版本，因为它看起来比较简洁，同时阐述了该行为的思想。利用这个版本可以编写出可维护性更好的代码，因为不需要复制代码，就可在两个独立的重载中执行相乘操作。

下一个要重载的乘法运算符支持矢量相乘。在数学领域，矢量相乘有两种方式，但这里我们感兴趣的是点积或内积，其结果实际上是一个标量。这就是我们介绍这个例子的原因：算术运算符不必返回与定义它们的类相同的类型。

在数学术语中，如果有两个矢量(x, y, z)和(X, Y, Z)，其内积就定义为 x * X + y * Y + z * Z 的值。两个矢量这样相乘很奇怪，但这实际上很有用，因为它可以用于计算各种其他的数。当然，如果要使用 Direct3D 或 DirectDraw 编写代码来显示复杂的 3D 图形，那么在计算对象放在屏幕上的什么位置时，中间常常需要编写代码来计算矢量的内积。这里我们关心的是使用 Vector 编写出 double X = a * b，其中 a 和 b 是两个 Vector 对象，并计算出它们的点积。相关的运算符重载如下所示：

```
public static double operator *(Vector left, Vector right) =>
  left.X * right.X + left.Y * right.Y + left.Z * right.Z;
```

理解了算术运算符后，就可以用一个简单的测试方法来检验它们是否能正常运行(代码文件 OperatorOverloadingSample2/Program.cs)：

```
static void Main()
{
  // stuff to demonstrate arithmetic operations
  Vector vect1, vect2, vect3;
  vect1 = new Vector(1.0, 1.5, 2.0);
  vect2 = new Vector(0.0, 0.0, -10.0);
  vect3 = vect1 + vect2;
  Console.WriteLine($"vect1 = {vect1}");
  Console.WriteLine($"vect2 = {vect2}");
  Console.WriteLine($"vect3 = vect1 + vect2 = {vect3}");
  Console.WriteLine($"2 * vect3 = {2 * vect3}");
  Console.WriteLine($"vect3 += vect2 gives {vect3 += vect2}");
  Console.WriteLine($"vect3 = vect1 * 2 gives {vect3 = vect1 * 2}");
  Console.WriteLine($"vect1 * vect3 = {vect1 * vect3}");
}
```

运行此代码，得到如下所示的结果：

```
vect1 = ( 1, 1.5, 2 )
vect2 = ( 0, 0, -10 )
vect3 = vect1 + vect2 = ( 1, 1.5, -8 )
2 * vect3 = ( 2, 3, -16 )
vect3 += vect2 gives ( 1, 1.5, -18 )
vect3 = vect1 * 2 gives ( 2, 3, 4 )
vect1 * vect3 = 14.5
```

这说明，运算符重载会给出正确的结果，但如果仔细看看测试代码，就会惊奇地注意到，实际上它使用的是没有重载的运算符——相加赋值运算符(+=)：

```
Console.WriteLine($"vect3 += vect2 gives {vect3 += vect2}");
```

虽然+=一般计为单个运算符，但实际上它对应的操作分为两步：相加和赋值。与 C++语言不同，C#不允许重载=运算符；但如果重载+运算符，编译器就会自动使用+运算符的重载来执行+=运算符的操作。－=、*=、/=和&=等所有赋值运算符也遵循此原则。

### 6.6.3　比较运算符的重载

本章前面介绍过，C#中有 6 个比较运算符，它们分为 3 对：

- ==和!=
- >和<
- >=和<=

> **注意：**
> .NET 指南指定，在比较两个对象时，如果==运算符返回 true，就应总是返回 true。所以应在不可改变的类型上只重载==运算符。

C#语言要求成对重载比较运算符。即，如果重载了==，也就必须重载!=，否则会产生编译器错误。另外，比较运算符必须返回布尔类型的值。这是它们与算术运算符的根本区别。例如，两个数相加或相减的结果理论上取决于这些数值的类型。前面提到，两个 Vector 对象相乘会得到一个标量。另一个例子是.NET 基类 System.DateTime。两个 DateTime 实例相减，得到的结果不是一个 DateTime，而是一个 System.TimeSpan 实例。相比之下，如果比较运算得到的不是布尔类型的值，就没有任何意义。

除了这些区别外，重载比较运算符所遵循的原则与重载算术运算符相同。但比较两个数并不那么简单。例如，如果只比较两个对象引用，就是比较存储对象的内存地址。比较运算符很少进行这样的比较，所以必须编写代码重载运算符，比较对象的值，并返回相应的布尔结果。下面对 Vector 结构重载==和!=运算符。首先是实现==重载的代码(代码文件 OverloadingComparisonSample/Vector.cs)：

```
public static bool operator ==(Vector left, Vector right)
{
  if (object.ReferenceEquals(left, right)) return true;
  return left.X == right.X && left.Y == right.Y && left.Z == right.Z;
}
```

这种方式仅根据 Vector 组成元素的值来对它们进行相等性比较。对于大多数结构，这就是我们希望的方式，但在某些情况下，可能需要仔细考虑相等性的含义。例如，如果有嵌入的类，那么是应比较引用是否指向同一个对象(浅度比较)，还是应比较对象的值是否相等(深度比较)？

浅度比较是比较对象是否指向内存中的同一个位置，而深度比较是比较对象的值和属性是否相等。应根据具体情况进行相等性检查，从而有助于确定要验证的结果。

> **注意：**
> 不要通过调用从 System.Object 中继承的 Equals()方法的实例版本来重载比较运算符。如果这么做，在 objA 是 null 时判断(objA==objB)，就会产生一个异常，因为.NET 运行库会试图判断 null.Equals(objB)。采用其他方法(重写 Equals()方法以调用比较运算符)比较安全。

还需要重载运算符!=，采用的方式如下：

```
public static bool operator !=(Vector left, Vector right) => !(left == right);
```

现在重写 Equals 和 GetHashCode 方法。这些方法应该总是在重写==运算符时进行重写，否则编译器会报错。

```
public override bool Equals(object obj)
{
  if (obj == null) return false;
  return this == (Vector)obj;
```

```
}

public override int GetHashCode() =>
  X.GetHashCode() ^ (Y.GetHashCode() ^ Z.GetHashCode();
```

Equals 方法可以转而调用==运算符。散列代码的实现应比较快速，且总是对相同的对象返回相同的值。这个方法在使用字典时很重要。在字典中，它用来建立对象的树，所以最好把返回值分布到整数范围内。double 类型的 GetHashCode 方法返回 double 值的整数表示。对于 Vector 类型，只是通过 XOR 合并底层类型的散列值。

对于值类型，也应该实现接口 IEquatable<T>。这个接口是 Equals 方法的一个强类型化版本，由基类 Object 定义。有了所有其他代码，就很容易实现该方法：

```
public bool Equals(Vector other) => this == other;
```

像往常一样，应该用一些测试代码快速检查重写方法的工作情况。这次定义 3 个 Vector 对象，并进行比较(代码文件 OverloadingComparisonSample/Program.cs)：

```
static void Main()
{
  var vect1 = new Vector(3.0, 3.0, -10.0);
  var vect2 = new Vector(3.0, 3.0, -10.0);
  var vect3 = new Vector(2.0, 3.0, 6.0);
  Console.WriteLine($"vect1 == vect2 returns {(vect1 == vect2)}");
  Console.WriteLine($"vect1 == vect3 returns {(vect1 == vect3)}");
  Console.WriteLine($"vect2 == vect3 returns {(vect2 == vect3)}");
  Console.WriteLine();
  Console.WriteLine($"vect1 != vect2 returns {(vect1 != vect2)}");
  Console.WriteLine($"vect1 != vect3 returns {(vect1 != vect3)}");
  Console.WriteLine($"vect2 != vect3 returns {(vect2 != vect3)}");
}
```

在命令行上运行该示例，生成如下结果：

```
vect1 == vect2 returns True
vect1 == vect3 returns False
vect2 == vect3 returns False
vect1 != vect2 returns False
vect1 != vect3 returns True
vect2 != vect3 returns True
```

### 6.6.4 可以重载的运算符

并不是所有的运算符都可以重载。可以重载的运算符如表 6-5 所示。

表 6-5

| 类 别 | 运 算 符 | 限 制 |
|---|---|---|
| 算术二元运算符 | +、*、/、–、% | 无 |
| 算术一元运算符 | +、–、++、–– | 无 |
| 按位二元运算符 | &、\|、^、<<、>> | 无 |
| 按位一元运算符 | !、~、true、false | true 和 false 运算符必须成对重载 |
| 比较运算符 | ==、!=、>=、<、<=> | 比较运算符必须成对重载 |
| 赋值运算符 | +=、–=、*=、/=、>>=、<<=、%=、&=、\|=、^= | 不能显式地重载这些运算符，在重写单个运算符(如+、–、%等)时，它们会被隐式地重写 |
| 索引运算符 | [] | 不能直接重载索引运算符。第 2 章介绍的索引器成员类型允许在类和结构上支持索引运算符 |
| 类型强制转换运算符 | () | 不能直接重载类型强制转换运算符。用户定义的类型强制转换(本章后面介绍)允许定义定制的类型强制转换行为 |

**注意：**

为什么要重载 true 和 false 操作符？有一个很好的原因：根据所使用的技术或框架，哪些整数值代表 true 或 false 是不同的。在许多技术中，0 是 false，1 是 true；其他技术把非 0 值定义为 true，还有一些技术把 −1 定义为 false。

## 6.7　实现自定义的索引运算符

自定义索引器不能使用运算符重载语法来实现，但是它们可以用与属性非常相似的语法来实现。

首先看看数组元素的访问。这里创建一个 int 元素数组。第二行代码使用索引器来访问第二个元素，并给它传递 42。第三行使用索引器来访问第三个元素，并给该元素传递变量 x。

```
int[] arr1 = {1, 2, 3};
arr1[1] = 42;
int x = arr1[2];
```

**注意：**

数组在第 7 章阐述。

CustomIndexerSample 使用如下名称空间：

```
System
System.Collections.Generic
System.Linq
```

要创建自定义索引器，首先要创建一个 Person 类，其中包含 FirstName、LastName 和 Birthday 只读属性(代码文件 CustomIndexerSample/Person.cs)：

```
public class Person
{
  public DateTime Birthday { get; }
  public string FirstName { get; }
  public string LastName { get; }

  public Person(string firstName, string lastName, DateTime birthDay)
  {
    FirstName = firstName;
    LastName = lastName;
    Birthday = birthDay;
  }

  public override string ToString() => $"{FirstName} {LastName}";
}
```

类 PersonCollection 定义了一个包含 Person 元素的私有数组字段，以及一个可以传递许多 Person 对象的构造函数(代码文件 CustomIndexerSample/PersonCollection.cs)：

```
public class PersonCollection
{
  private Person[] _people;
  public PersonCollection(params Person[] people) =>
    _people = people.ToArray();
}
```

为了允许使用索引器语法访问 PersonCollection 并返回 Person 对象，可以创建一个索引器。索引器看起来非常类似于属性，因为它也包含 get 和 set 访问器。两者的不同之处是名称。指定索引器要使用 this 关键字。this 关键字后面的括号指定索引使用的类型。数组提供 int 类型的索引器，所以这里使用 int 类型直接把信息传递给被包含的数组 people。get 和 set 访问器的使用非常类似于属性。检索值时调用 get 访问器，在右边传递 Person 对象时调用 set 访问器。

```
public Person this[int index]
{
  get => _people[index];
  set => _people[index] = value;
}
```

对于索引器，不能仅定义 int 类型作为索引类型。任何类型都是有效的，如下面的代码所示，其中把 DateTime 结构作为索引类型。这个索引器用来返回有指定生日的每个人。因为多个人员可以有相同的生日，所以不是返回一个 Person 对象，而是用接口 IEnumerable<Person>返回一个 Person 对象列表。所使用的 Where 方法根据 lambda 表达式进行过滤。Where 方法在名称空间 System.Linq 中定义：

```
public IEnumerable<Person> this[DateTime birthDay]
{
  get => _people.Where(p => p.Birthday == birthDay);
}
```

使用 DateTime 类型的索引器检索人员对象，但不允许把人员对象设置为只有 get 访问器，而没有 set 访问器。在 C# 6 中有一个速记符号，可使用表达式主体的成员创建相同的代码(属性也可使用该语法)：

```
public IEnumerable<Person> this[DateTime birthDay] =>
  _people.Where(p => p.Birthday == birthDay);
```

示例应用程序的 Main()方法创建一个 PersonCollection 对象，给构造函数传递 4 个 Person 对象。在第一个 WriteLine 方法中，使用索引器的 get 访问器和 int 参数访问第三个元素。在 foreach 循环中，带有 DateTime 参数的索引器用来传递指定的日期(代码文件 CustomIndexerSample/Program.cs)：

```
static void Main()
{
  var p1 = new Person("Ayrton", "Senna", new DateTime(1960, 3, 21));
  var p2 = new Person("Ronnie", "Peterson", new DateTime(1944, 2, 14));
  var p3 = new Person("Jochen", "Rindt", new DateTime(1942, 4, 18));
  var p4 = new Person("Francois", "Cevert", new DateTime(1944, 2, 25));
  var coll = new PersonCollection(p1, p2, p3, p4);
  Console.WriteLine(coll[2]);
  foreach (var r in coll[new DateTime(1960, 3, 21)])
  {
    Console.WriteLine(r);
  }
  Console.ReadLine();
}
```

运行程序，第一个 WriteLine 方法把 Jochen Rindt 写到控制台。foreach 循环的结果是 Ayrton Senna，因为他的生日是第二个索引器中指定的日期。

## 6.8 用户定义的类型强制转换

本章前面介绍了如何在预定义的数据类型之间转换数值，这通过类型强制转换过程来完成。C#允许进行两种不同类型的强制转换：隐式强制转换和显式强制转换。本节将讨论这两种类型的强制转换。

显式强制转换要在代码中显式地标记强制转换，即应该在圆括号中写出目标数据类型：

```
int i = 3;
long l = i; // implicit
short s = (short)i; // explicit
```

对于预定义的数据类型，当类型强制转换可能失败或丢失某些数据时，需要显式强制转换。例如：

- 把 int 转换为 short 时，short 可能不够大，不能包含对应 int 的数值。
- 把有符号的数据类型转换为无符号的数据类型时，如果有符号的变量包含一个负值，就会得到不正确的结果。
- 把浮点数转换为整数数据类型时，数字的小数部分会丢失。
- 把可空类型转换为非可空类型时，null 值会导致异常。

此时应在代码中进行显式强制转换，告诉编译器你知道存在丢失数据的危险，因此编写代码时要把这种可能性考虑在内。

C#允许定义自己的数据类型(结构和类)，这意味着需要某些工具支持在自定义的数据类型之间进行类型强制转换。方法是把类型强制转换运算符定义为相关类的一个成员运算符。类型强制转换运算符必须标记为隐式或显式，以说明希望如何使用它。我们应遵循与预定义的类型强制转换相同的指导原则；如果知道无论在源变

量中存储什么值，类型强制转换总是安全的，就可以把它定义为隐式强制转换。然而，如果某些数值可能会出错，如丢失数据或抛出异常，就应把数据类型转换定义为显式强制转换。

> **注意：**
> 如果源数据值会使类型强制转换失败，或者可能会抛出异常，就应把任何自定义类型强制转换定义为显式强制转换。

定义类型强制转换的语法类似于本章前面介绍的重载运算符。这并不是偶然现象，类型强制转换在某种情况下可以看为一种运算符，其作用是从源类型转换为目标类型。为了说明这种语法，下面的代码从本节后面介绍的结构 Currency 示例中节选而来：

```
public static implicit operator float (Currency value)
{
  // processing
}
```

运算符的返回类型定义了类型强制转换操作的目标类型，它有一个参数，即要转换的源对象。这里定义的类型强制转换可以隐式地把 Currency 型的值转换为 float 型。注意，如果数据类型转换声明为隐式，编译器就可以隐式或显式地使用这个转换。如果数据类型转换声明为显式，编译器就只能显式地使用它。与其他运算符重载一样，类型强制转换必须同时声明为 public 和 static。

> **注意：**
> C++开发人员应注意，这种情况与 C++中的用法不同，在 C++中，类型强制转换用于类的实例成员。

### 6.8.1　实现用户定义的类型强制转换

本节将在示例 SimpleCurrency 中介绍隐式和显式的用户定义类型强制转换用法。在这个示例中，定义一个结构 Currency，它包含一个正的 USD($)金额。C#为此提供了 decimal 类型，但如果要进行比较复杂的财务处理，仍可以编写自己的结构和类来表示相应的金额，在这样的类上实现特定的方法。

> **注意：**
> 类型强制转换的语法对于结构和类是一样的。本示例定义了一个结构，但把 Currency 声明为类也是可行的。

首先，Currency 结构的定义如下所示(代码文件 CastingSample/Currency.cs)：

```
public struct Currency
{
  public uint Dollars { get; }
  public ushort Cents { get; }

  public Currency(uint dollars, ushort cents)
  {
    Dollars = dollars;
    Cents = cents;
  }

  public override string ToString() => $"${Dollars}.{Cents,-2:00}";
}
```

Dollars 和 Cents 属性使用无符号的数据类型，可以确保 Currency 实例只能包含正值。采用这样的限制是为了在后面说明显式强制转换的一些要点。可以像这样使用一个类来存储公司员工的薪水信息。员工的薪水不会是负值！

下面先假定要把 Currency 实例转换为 float 值，其中 float 值的整数部分表示美元。换言之，应编写下面的代码：

```
var balance = new Currency(10, 50);
float f = balance; // We want f to be set to 10.5
```

为此，需要定义一种类型强制转换。给 Currency 的定义添加下述代码：

```
public static implicit operator float (Currency value) =>
value.Dollars + (value.Cents/100.0f);
```

这种类型强制转换是隐式的。在本例中这是一种合理的选择，因为在 Currency 的定义中，可以存储在 Currency 中的值也都可以存储在 float 数据中。在这种强制转换中，不应出现任何错误。

**注意：**

这里有一点欺骗性：实际上，当把 uint 转换为 float 时，精确度会降低，但 Microsoft 认为这种错误并不重要，因此把从 uint 到 float 的类型强制转换都当作隐式转换。

但是，如果把 float 型转换为 Currency 型，就不能保证转换肯定成功了。float 型可以存储负值，而 Currency 实例不能，且 float 型存储数值的数量级要比 Currency 型的(uint)Dollar 字段大得多。所以，如果 float 型包含一个不合适的值，把它转换为 Currency 型就会得到意想不到的结果。因此，从 float 型转换到 Currency 型就应定义为显式转换。下面是我们的第一次尝试，这次不会得到正确的结果，但有助于解释原因：

```
public static explicit operator Currency (float value)
{
  uint dollars = (uint)value;
  ushort cents = (ushort)((value-dollars)*100);
  return new Currency(dollars, cents);
}
```

下面的代码现在可以成功编译：

```
float amount = 45.63f;
Currency amount2 = (Currency)amount;
```

但是，下面的代码会抛出一个编译错误，因为它试图隐式地使用一个显式的类型强制转换：

```
float amount = 45.63f;
Currency amount2 = amount; // wrong
```

把类型强制转换声明为显式，就是警告开发人员要小心，因为可能会丢失数据。但这不是我们希望的 Currency 结构的行为方式。下面编写一个测试程序，并运行该示例。其中有一个 Main()方法，它实例化一个 Currency 结构，并试图进行几次转换。在这段代码的开头，以两种不同的方式计算 balance 的值，因为要使用它们来说明后面的内容(代码文件 CastingSample/Program.cs)：

```
static void Main()
{
  try
  {
    var balance = new Currency(50,35);
    Console.WriteLine(balance);
    Console.WriteLine($"balance is {balance}"); // implicitly invokes ToString
    float balance2 = balance;
    Console.WriteLine($"After converting to float, = {balance2}");
    balance = (Currency) balance2;
    Console.WriteLine($"After converting back to Currency, = {balance}");
    Console.WriteLine("Now attempt to convert out of range value of " +
      "-$50.50 to a Currency:");

    checked
    {
      balance = (Currency) (-50.50);
      Console.WriteLine($"Result is {balance}");
    }
  }
  catch(Exception e)
  {
    Console.WriteLine($"Exception occurred: {e.Message}");
  }
}
```

注意，所有的代码都放在一个 try 块中，以捕获在类型强制转换过程中发生的任何异常。在 checked 块中还添加了把超出范围的值转换为 Currency 的测试代码，以试图捕获负值。运行这段代码，得到如下所示的结果：

```
50.35
Balance is $50.35
After converting to float, = 50.35
After converting back to Currency, = $50.34
Now attempt to convert out of range value of -$50.50 to a Currency:
Result is $4294967246.00
```

这个结果表示代码并没有像我们希望的那样工作。首先，从 float 型转换回 Currency 型得到一个错误的结果$50.34，而不是$50.35。其次，在试图转换明显超出范围的值时，没有生成异常。

第一个问题是由舍入错误引起的。如果类型强制转换用于把 float 值转换为 uint 值，计算机就会截去多余的数字，而不是执行四舍五入。计算机以二进制而非十进制方式存储数字，小数部分 0.35 不能用二进制小数来精确表示(像 1/3 这样的分数不能精确地表示为十进制小数，它应等于循环小数 0.3333)。所以，计算机最后存储了一个略小于 0.35 的值，它可以用二进制格式精确地表示。把该数字乘以 100，就会得到一个小于 35 的数字，它截去了 34 美分。显然在本例中，这种由截去引起的错误是很严重的，避免该错误的方式是确保在数字转换过程中执行智能的四舍五入操作。

幸运的是，Microsoft 编写了一个类 System.Convert 来完成该任务。System.Convert 对象包含大量的静态方法来完成各种数字转换，我们需要使用的是 Convert.ToUInt16()。注意，在使用 System.Convert 类的方法时会造成额外的性能损失，所以只应在需要时使用它们。

下面看看为什么没有抛出期望的溢出异常。此处的问题是溢出异常实际发生的位置根本不在 Main()例程中——它是在强制转换运算符的代码中发生的，该代码在 Main()方法中调用，而且没有标记为 checked。

其解决方法是确保类型强制转换本身也在 checked 环境下进行。进行了这两处修改后，修订的转换代码如下所示。

```
public static explicit operator Currency (float value)
{
  checked
  {
    uint dollars = (uint)value;
    ushort cents = Convert.ToUInt16((value-dollars)*100);
    return new Currency(dollars, cents);
  }
```

注意，使用 Convert.ToUInt16()计算数字的美分部分，如上所示，但没有使用它计算数字的美元部分。在计算美元值时不需要使用 System.Convert，因为在此我们希望截去 float 值。

**注意：**

System.Convert 类的方法还执行它们自己的溢出检查。因此对于本例的情况，不需要把对 Convert.ToUInt16()的调用放在 checked 环境下。但把 value 显式地强制转换为美元值仍需要 checked 环境。

这里没有给出这个新的 checked 强制转换的结果，因为在本节后面还要对 SimpleCurrency 示例进行一些修改。

**注意：**

如果定义了一种使用非常频繁的类型强制转换，其性能也非常好，就可以不进行任何错误检查。如果对用户定义的类型强制转换和没有检查的错误进行了清晰的说明，这也是一种合理的解决方案。

### 1. 类之间的类型强制转换

Currency 示例仅涉及与 float(一种预定义的数据类型)来回转换的类。但类型转换不一定会涉及任何简单的数据类型。定义不同结构或类的实例之间的类型强制转换是完全合法的，但有两点限制：

- 如果某个类派生自另一个类，就不能定义这两个类之间的类型强制转换(这些类型的强制转换已经存在)。
- 类型强制转换必须在源数据类型或目标数据类型的内部定义。

为说明这些要求，假定有如图 6-1 所示的类层次结构。

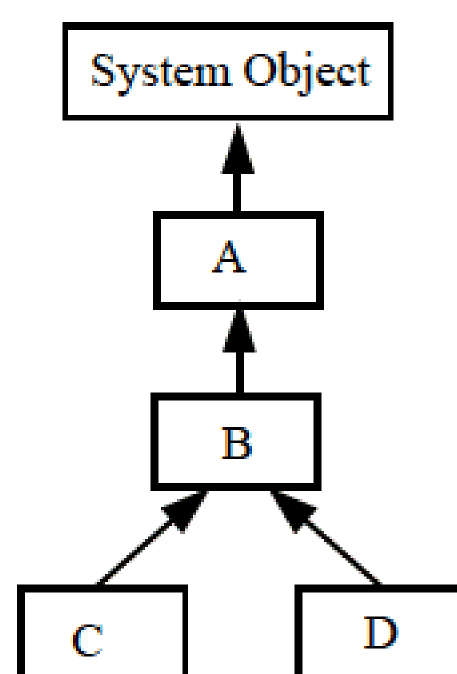



图 6-1

换言之，类 C 和 D 间接派生于 A。在这种情况下，在 A、B、C 或 D 之间唯一合法的自定义类型强制转换就是类 C 和 D 之间的转换，因为这些类并没有互相派生。对应的代码如下所示(假定希望类型强制转换是显式的，这是在用户定义的类之间定义类型强制转换的通常情况)：

```
public static explicit operator D(C value)
{
  //...
}

public static explicit operator C(D value)
{
  //...
}
```

对于这些类型强制转换，可以选择放置定义的地方——在 C 的类定义内部，或者在 D 的类定义内部，但不能在其他地方定义。C#要求把类型强制转换的定义放在源类(或结构)或目标类(或结构)的内部。这一要求的副作用是不能定义两个类之间的类型强制转换，除非至少可以编辑其中一个类的源代码。这是因为，这样可以防止第三方把类型强制转换引入类中。

一旦在一个类的内部定义了类型强制转换，就不能在另一个类中定义相同的类型强制转换。显然，对于每一种转换只能有一种类型强制转换，否则编译器就不知道该选择哪个类型强制转换。

### 2. 基类和派生类之间的类型强制转换

要了解这些类型强制转换是如何工作的，首先看看源和目标数据类型都是引用类型的情况。考虑两个类 MyBase 和 MyDerived，其中 MyDerived 直接或间接派生自 MyBase。

首先是从 MyDerived 到 MyBase 的转换，代码如下(假定提供了构造函数)：

```
MyDerived derivedObject = new MyDerived();
MyBase baseCopy = derivedObject;
```

在本例中，是从 MyDerived 隐式地强制转换为 MyBase。这是可行的，因为对类型 MyBase 的任何引用都可以引用 MyBase 类的对象或派生自 MyBase 的对象。在 OO 编程中，派生类的实例实际上是基类的实例，但加入了一些额外的信息。在基类上定义的所有函数和字段也都在派生类上得到定义。

下面分析另一种方式，编写如下的代码：

```
MyBase derivedObject = new MyDerived();
MyBase baseObject = new MyBase();
MyDerived derivedCopy1 = (MyDerived) derivedObject; // OK
MyDerived derivedCopy2 = (MyDerived) baseObject; // Throws exception
```

上面的代码都是合法的 C#代码(从语法的角度来看是合法的)，它说明了把基类强制转换为派生类。但是，在执行时最后一条语句会抛出一个异常。在进行类型强制转换时，会检查被引用的对象。因为基类引用原则上可以引用一个派生类的实例，所以这个对象可能是要强制转换的派生类的一个实例。如果是这样，强制转换就会成功，派生的引用设置为引用这个对象。但如果该对象不是派生类(或者派生于这个类的其他类)的一个实例，强制转换就会失败，并抛出一个异常。

注意，编译器已经提供了基类和派生类之间的强制转换，这种转换实际上并没有对讨论的对象进行任何数据转换。如果要进行的转换是合法的，它们也仅是把新引用设置为对对象的引用。这些强制转换在本质上与用户定义的强制转换不同。例如，在前面的 SimpleCurrency 示例中，我们定义了 Currency 结构和 float 数之间的强制转换。在 float 型到 Currency 型的强制转换中，实际上实例化了一个新的 Currency 结构，并用要求的值初始化它。在基类和派生类之间的预定义强制转换则不是这样。如果实际上要把 MyBase 实例转换为真实的 MyDerived 对象，该对象的值根据 MyBase 实例的内容来确定，就不能使用类型强制转换语法。最合适的选项通常是定义一个派生类的构造函数，它以基类的实例作为参数，让这个构造函数完成相关的初始化：

```
class DerivedClass: BaseClass
{
  public DerivedClass(BaseClass base)
  {
    // initialize object from the Base instance
  }
  // ...
```

### 3. 装箱和拆箱类型强制转换

前面主要讨论了基类和派生类之间的类型强制转换，其中，基类和派生类都是引用类型。类似的原则也适用于强制转换值类型，尽管在转换值类型时，不可能仅仅复制引用，还必须复制一些数据。

当然，不能从结构或基本值类型中派生。所以基本结构和派生结构之间的强制转换总是基本类型或结构与 System.Object 之间的转换(理论上可以在结构和 System.ValueType 之间进行强制转换，但一般很少这么做)。

从结构(或基本类型)到 object 的强制转换总是一种隐式的强制转换，因为这种强制转换是从派生类型到基本类型的转换，即第 2 章简要介绍的装箱过程。例如，使用 Currency 结构：

```
var balance = new Currency(40,0);
object baseCopy = balance;
```

在执行上述隐式的强制转换时，balance的内容被复制到堆上，放在一个装箱的对象中，并且 baseCopy 对象引用被设置为该对象。在后台实际发生的情况是：在最初定义 Currency 结构时，.NET Framework 隐式地提供另一个(隐藏的)类，即装箱的 Currency 类，它包含与 Currency 结构相同的所有字段，但它是一个引用类型，存储在堆上。无论定义的这个值类型是一个结构，还是一个枚举，定义它时都存在类似的装箱引用类型，对应于所有的基本值类型，如 int、double 和 uint 等。不能也不必在源代码中直接通过编程访问某些装箱类，但在把一个值类型强制转换为 object 型时，它们是在后台工作的对象。在隐式地把 Currency 转换为 object 时，会实例化一个装箱的 Currency 实例，并用 Currency 结构中的所有数据进行初始化。在上面的代码中，baseCopy 对象引用的就是这个已装箱的 Currency 实例。通过这种方式，就可以实现从派生类型到基本类型的强制转换，并且值类型的语法与引用类型的语法一样。

强制转换的另一种方式称为拆箱。与在基本引用类型和派生引用类型之间的强制转换一样，这是一种显式的强制转换，因为如果要强制转换的对象不是正确的类型，就会抛出一个异常：

```
object derivedObject = new Currency(40,0);
object baseObject = new object();
Currency derivedCopy1 = (Currency)derivedObject; // OK
Currency derivedCopy2 = (Currency)baseObject; // Exception thrown
```

上述代码的工作方式与前面关于引用类型的代码一样。把 derivedObject 强制转换为 Currency 会成功执行，因为 derivedObject 实际上引用的是装箱 Currency 实例——强制转换的过程是把已装箱的 Currency 对象的字段复制到一个新的 Currency 结构中。第二种强制转换会失败，因为 baseObject 没有引用已装箱的 Currency 对象。

在使用装箱和拆箱时，这两个过程都把数据复制到新装箱或拆箱的对象上，理解这一点非常重要。这样，对装箱对象的操作就不会影响原始值类型的内容。

## 6.8.2　多重类型强制转换

在定义类型强制转换时必须考虑的一个问题是，如果在进行要求的数据类型转换时没有可用的直接强制转换方式，C#编译器就会寻找一种转换方式，把几种强制转换合并起来。例如，在 Currency 结构中，假定编译器

遇到下面几行代码：

```
var balance = new Currency(10,50);
long amount = (long)balance;
double amountD = balance;
```

首先初始化一个 Currency 实例，再把它转换为 long 型。问题是没有定义这样的强制转换。但是，这段代码仍可以编译成功。因为编译器知道我们已经定义一个从 Currency 到 float 的隐式强制转换，而且它知道如何显式地从 float 强制转换为 long。所以它会把这行代码编译为中间语言(IL)代码，IL 代码首先把 balance 转换为 float 型，再把结果转换为 long 型。把 balance 转换为 double 型时，在上述代码的最后一行中也执行了同样的操作。因为从 Currency 到 float 的强制转换和从 float 到 double 的预定义强制转换都是隐式的，所以可以在编写代码时把这种转换当作一种隐式转换。如果要显式地指定强制转换过程，则可以编写如下代码：

```
var balance = new Currency(10,50);
long amount = (long)(float)balance;
double amountD = (double)(float)balance;
```

但是在大多数情况下，这会使代码变得比较复杂，因此是不必要的。相比之下，下面的代码会产生一个编译错误：

```
var balance = new Currency(10,50);
long amount = balance;
```

原因是编译器可以找到的最佳匹配转换仍是首先转换为 float 型，再转换为 long 型。但需要显式地指定从 float 型到 long 型的转换。

并非所有这些转换都会带来太多的麻烦。毕竟转换的规则非常直观，主要是为了防止在开发人员不知情的情况下丢失数据。但是，在定义类型强制转换时如果不小心，编译器就有可能指定一条导致不期望结果的路径。例如，假定编写 Currency 结构的其他小组成员要把一个 uint 数据转换为 Currency 型，其中该 uint 数据中包含了美分的总数(是美分而非美元，因为我们不希望丢失美元的小数部分)。为此应编写如下代码来实现强制转换：

```
// Do not do this!
public static implicit operator Currency (uint value) =>
  new Currency(value/100u, (ushort)(value%100));
```

注意，在这段代码中，第一个 100 后面的 u 可以确保把 value/100u 解释为一个 uint 值。如果写成 value/100，编译器就会把它解释为一个 int 型的值，而不是 uint 型的值。

在这段代码中清楚地标注了“Do not do it(不要这么做)”。下面说明其原因。看看下面的代码段，它把包含值 350 的一个 uint 数据转换为一个 Currency，再转换回 uint 型。那么在执行完这段代码后，bal2 中又将包含什么？

```
uint bal = 350;
Currency balance = bal;
uint bal2 = (uint)balance;
```

答案不是 350，而是 3！而且这是符合逻辑的。我们把 350 隐式地转换为 Currency，得到的结果是 balance.Dollars=3 和 balance.Cents=50。然后编译器进行通常的操作，为转换回 uint 型指定最佳路径。balance 最终会被隐式地转换为 float 型(其值为 3.5)，然后显式地转换为 uint 型，其值为 3。

当然，在其他示例中，转换为另一种数据类型后，再转换回来有时会丢失数据。例如，把包含 5.8 的 float 数值转换为 int 数值，再转换回 float 数值，会丢失数字中的小数部分，得到 5，但原则上，丢失数字的小数部分和一个整数被大于 100 的数整除的情况略有区别。Currency 现在成为一种相当危险的类，它会对整数进行一些奇怪的操作。

问题是，在转换过程中如何解释整数存在冲突。从 Currency 型到 float 型的强制转换会把整数 1 解释为 1 美元，但从 uint 型到 Currency 型的强制转换会把这个整数解释为 1 美分，这是很糟糕的一个示例。如果希望类易于使用，就应确保所有的强制转换都按一种互相兼容的方式执行，即这些转换直观上应得到相同的结果。在本例中，显然要重新编写从 uint 型到 Currency 型的强制转换，把整数值 1 解释为 1 美元：

```
public static implicit operator Currency (uint value) =>
  new Currency(value, 0);
```

偶尔你也会觉得这种新的转换方式可能根本不必要。但实际上，这种转换方式可能非常有用。没有这种强

制转换，编译器在执行从 uint 型到 Currency 型的转换时，就只能通过 float 型来进行。此时直接转换的效率要高得多，所以进行这种额外的强制转换会提高性能，但需要确保它的结果与通过 float 型进行转换得到的结果相同。在其他情况下，也可以为不同的预定义数据类型分别定义强制转换，让更多的转换隐式地执行，而不是显式地执行，但本例不是这样。

测试这种强制转换是否兼容，应确定无论使用什么转换路径，它是否都能得到相同的结果(而不是像在从 float 型到 int 型的转换过程中那样丢失数据)。Currency 类就是一个很好的示例。看看下面的代码：

```
var balance = new Currency(50, 35);
ulong bal = (ulong) balance;
```

目前，编译器只能采用一种方式来完成这个转换：把 Currency 型隐式地转换为 float 型，再显式地转换为 ulong 型。从 float 型到 ulong 型的转换需要显式转换，本例就显式指定了这个转换，所以编译是成功的。

但假定要添加另一种强制转换，从 Currency 型隐式地转换为 uint 型，就需要修改 Currency 结构，添加从 uint 型到 Currency 型的强制转换和从 Currency 型到 uint 型的强制转换 (代码文件 CastingSample/Currency.cs)：

```
public static implicit operator Currency (uint value) =>
  new Currency(value, 0);
public static implicit operator uint (Currency value) => value.Dollars;
```

现在，编译器从 Currency 型转换到 ulong 型可以使用另一条路径：先从 Currency 型隐式地转换为 uint 型，再隐式地转换为 ulong 型。该采用哪条路径？ C#有一些严格的规则(本书不详细讨论这些规则，有兴趣的读者可参阅 MSDN 文档)，告诉编译器如何确定哪条是最佳路径。但最好自己设计类型强制转换，让所有的转换路径都得到相同的结果(但没有精确度的损失)，此时编译器选择哪条路径就不重要了(在本例中，编译器会选择 Currency→uint→ulong 路径，而不是 Currency→float→ulong 路径)。

为了测试把 Currency 强制转换为 uint 的过程，给 Main()方法添加如下代码(代码文件 CastingSample/Program.cs)：

```
static void Main()
{
  try
  {
    var balance = new Currency(50,35);
    Console.WriteLine(balance);
    Console.WriteLine($"balance is {balance}");
    uint balance3 = (uint) balance;
    Console.WriteLine($"Converting to uint gives {balance3}");
  }
  catch (Exception ex)
  {
    Console.WriteLine($"Exception occurred: {e.Message}");
  }
}
```

运行这个示例，得到如下所示的结果：

```
50
balance is $50.35
Converting to uint gives 50
```

这个结果显示了到 uint 型的转换是成功的，但在转换过程中丢失了 Currency 的美分部分(小数部分)。把负的 float 类型强制转换为 Currency 型也产生了预料中的溢出异常，因为 float 型到 Currency 型的强制转换本身定义了一个 checked 环境。

但是，这个输出结果也说明了进行强制转换时最后一个要注意的潜在问题：结果的第一行没有正确显示余额，显示了 50，而不是$50.35。

这是为什么？问题是在把类型强制转换和方法重载合并起来时，会出现另一个不希望的错误源。

WriteLine()语句使用格式字符串隐式地调用 Currency.ToString()方法，以确保 Currency 显示为一个字符串。

但是，第 1 行的 WriteLine()方法只把原始 Currency 结构传递给 WriteLine()。目前 WriteLine()有许多重载版本，但它们的参数都不是 Currency 结构。所以编译器会到处搜索，看看它能把 Currency 强制转换为什么类型，以便与 WriteLine()的一个重载方法匹配。如上所示，WriteLine()的一个重载方法可以快速而高效地显示 uint 型，

且其参数是一个 uint 值。因此应把 Currency 隐式地强制转换为 uint 型。

实际上，WriteLine()有另一个重载方法，它的参数是一个 double 值，结果显示该 double 的值。如果仔细看看前面 SimpleCurrency 示例的结果，就会发现该结果的第 1 行就是使用这个重载方法把 Currency 显示为 double 型。在这个示例中，没有直接把 Currency 强制转换为 uint 型，所以编译器选择 Currency→float→double 作为可用于 WriteLine()重载方法的首选强制转换方式。但在 SimpleCurrency2 中可以直接强制转换为 uint 型，所以编译器会选择该路径。

结论是：如果方法调用带有多个重载方法，并要给该方法传送参数，而该参数的数据类型不匹配任何重载方法，就可以迫使编译器确定使用哪些强制转换方式进行数据转换，从而决定使用哪个重载方法(并进行相应的数据转换)。当然，编译器总是按逻辑和严格的规则来工作，但结果可能并不是我们所期望的。如果存在任何疑问，最好指定显式地使用哪种强制转换。

## 6.9　小结

本章介绍了 C#提供的标准运算符，描述了对象的相等性机制，讨论了编译器如何将一种标准数据类型转换为另一种标准数据类型。本章还阐述了如何使用运算符重载在自己的数据类型上实现自定义运算符。最后，讨论了运算符重载的一种特殊类型，即类型强制转换运算符，它允许用户指定如何将自定义类型的实例转换为其他数据类型。

第 7 章将介绍数组，其中索引运算符有很重要的作用。

# 第 7 章

# 数　组

**本章要点**

- 简单数组
- 多维数组
- 锯齿数组
- Array 类
- 作为参数的数组
- 枚举
- 结构比较
- Span
- 数组池

**本章源代码下载地址(wrox.com)：**

打开 www.wrox.com 的 Download Code 选项卡可下载本章源代码。源代码也可以在 Arrays 目录的 https://github.com/ProfessionalCSharp/ProfessionalCSharp7 中找到。本章代码分为以下几个主要的示例文件：

- SimpleArrays
- SortingSample
- ArraySegment
- YieldSample
- StructuralComparison
- SpanSample
- ArrayPoolSample

## 7.1　相同类型的多个对象

如果需要使用相同类型的多个对象，就可以使用集合(参见第 10 章)和数组。C#用特殊的记号声明、初始化和使用数组。Array 类在后台发挥作用，它为数组中元素的排序和过滤提供了几个方法。使用枚举器，可以迭代数组中的所有元素。

> **注意：**
> 如果需要使用不同类型的多个对象，可以通过类、结构和元组使用它们。类和结构参见第 3 章，元组参见第 13 章。

# 7.2 简单数组

如果需要使用同一类型的多个对象，就可以使用数组。数组是一种数据结构，它可以包含同一类型的多个元素。

## 7.2.1 数组的声明

在声明数组时，应先定义数组中元素的类型，其后是一对空方括号和一个变量名。例如，下面声明了一个包含整型元素的数组：

```
int[] myArray;
```

## 7.2.2 数组的初始化

声明了数组后，就必须为数组分配内存，以保存数组的所有元素。数组是引用类型，所以必须给它分配堆上的内存。为此，应使用 new 运算符，指定数组中元素的类型和数量来初始化数组的变量。下面指定了数组的大小。

```
myArray = new int[4];
```

> **注意：**
> 值类型和引用类型请参见第 3 章。

在声明和初始化数组后，变量 myArray 就引用了 4 个整型值，它们位于托管堆上，如图 7-1 所示。

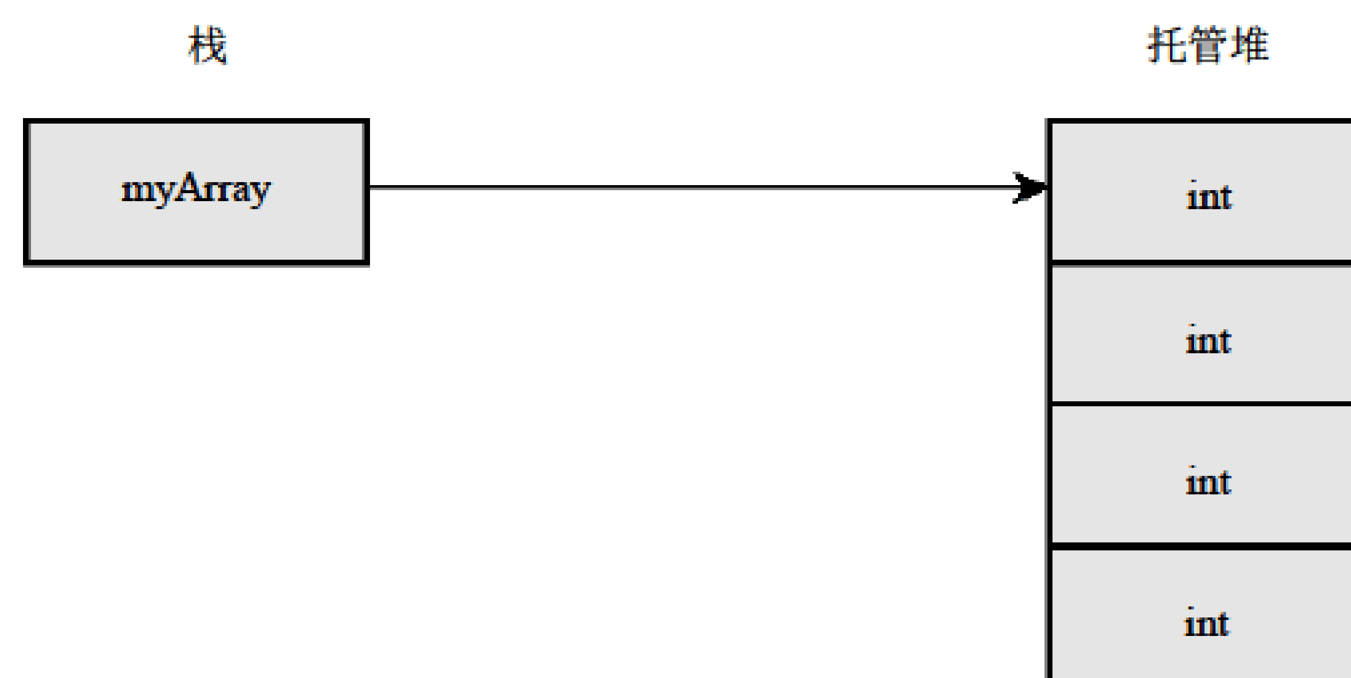



图 7-1

> **注意：**
> 在指定了数组的大小后，如果不复制数组中的所有元素，就不能重新设置数组的大小。如果事先不知道数组中应包含多少个元素，就可以使用集合(参见第 10 章)。

除了在两条语句中声明和初始化数组外，还可以在一条语句中声明和初始化数组：

```
int[] myArray = new int[4];
```

还可以使用数组初始化器为数组的每个元素赋值。数组初始化器只能在声明数组变量时使用，不能在声明数组之后使用。

```
int[] myArray = new int[4] {4, 7, 11, 2};
```

如果用花括号初始化数组，则还可以不指定数组的大小，因为编译器会自动统计元素的个数：

```
int[] myArray = new int[] {4, 7, 11, 2};
```

使用 C#编译器还有一种更简化的形式。使用花括号可以同时声明和初始化数组，编译器生成的代码与前面的例子相同：

```
int[] myArray = {4, 7, 11, 2};
```

### 7.2.3　访问数组元素

在声明和初始化数组后，就可以使用索引器访问其中的元素了。数组只支持有整型参数的索引器。

通过索引器传递元素编号，就可以访问数组。索引器总是以 0 开头，表示第一个元素。可以传递给索引器的最大值是元素个数减 1，因为索引从 0 开始。在下面的例子中，数组 myArray 用 4 个整型值声明和初始化。用索引器对应的值 0、1、2 和 3 就可以访问该数组中的元素。

```
int[] myArray = new int[] {4, 7, 11, 2};
int v1 = myArray[0]; // read first element
int v2 = myArray[1]; // read second element
myArray[3] = 44; // change fourth element
```

**注意：**

如果使用错误的索引器值(大于数组的长度)，就会抛出 IndexOutOfRangeException 类型的异常。

如果不知道数组中的元素个数，则可在 for 语句中使用 Length 属性：

```
for (int i = 0; i < myArray.Length; i++)
{
  Console.WriteLine(myArray[i]);
}
```

除了使用 for 语句迭代数组中的所有元素之外，还可以使用 foreach 语句：

```
foreach (var val in myArray)
{
  Console.WriteLine(val);
}
```

**注意：**

foreach 语句利用了本章后面讨论的 IEnumerable 和 IEnumerator 接口，从第一个索引遍历数组，直到最后一个索引。

### 7.2.4　使用引用类型

除了能声明预定义类型的数组，还可以声明自定义类型的数组。下面用 Person 类来说明，这个类有自动实现的只读属性 Firstname 和 Lastname，以及从 Object 类重写的 ToString()方法(代码文件 SimpleArrays/Person.cs)：

```
public class Person
{
  public Person(string firstName, string lastName)
  {
    FirstName = firstName;
    LastName = lastName;
  }

  public string FirstName { get; }
  public string LastName { get; }
  public override string ToString() => $"{FirstName} {LastName}";
}
```

声明一个包含两个 Person 元素的数组与声明一个 int 数组类似：

```
Person[] myPersons = new Person[2];
```

但是必须注意，如果数组中的元素是引用类型，就必须为每个数组元素分配内存。如果使用了数组中未分配内存的元素，则会抛出 NullReferenceException 类型的异常。

**注意：**
第 14 章介绍了错误和异常的详细内容。

使用从 0 开始的索引器，可以为数组的每个元素分配内存：

```
myPersons[0] = new Person("Ayrton", "Senna");
myPersons[1] = new Person("Michael", "Schumacher");
```

图 7-2 显示了 Person 数组中的对象在托管堆中的情况。myPersons 是存储在栈上的一个变量，该变量引用了存储在托管堆上的 Person 元素对应的数组。这个数组有足够容纳两个引用的空间。数组中的每一项都引用了一个 Person 对象，而这些 Person 对象也存储在托管堆上。

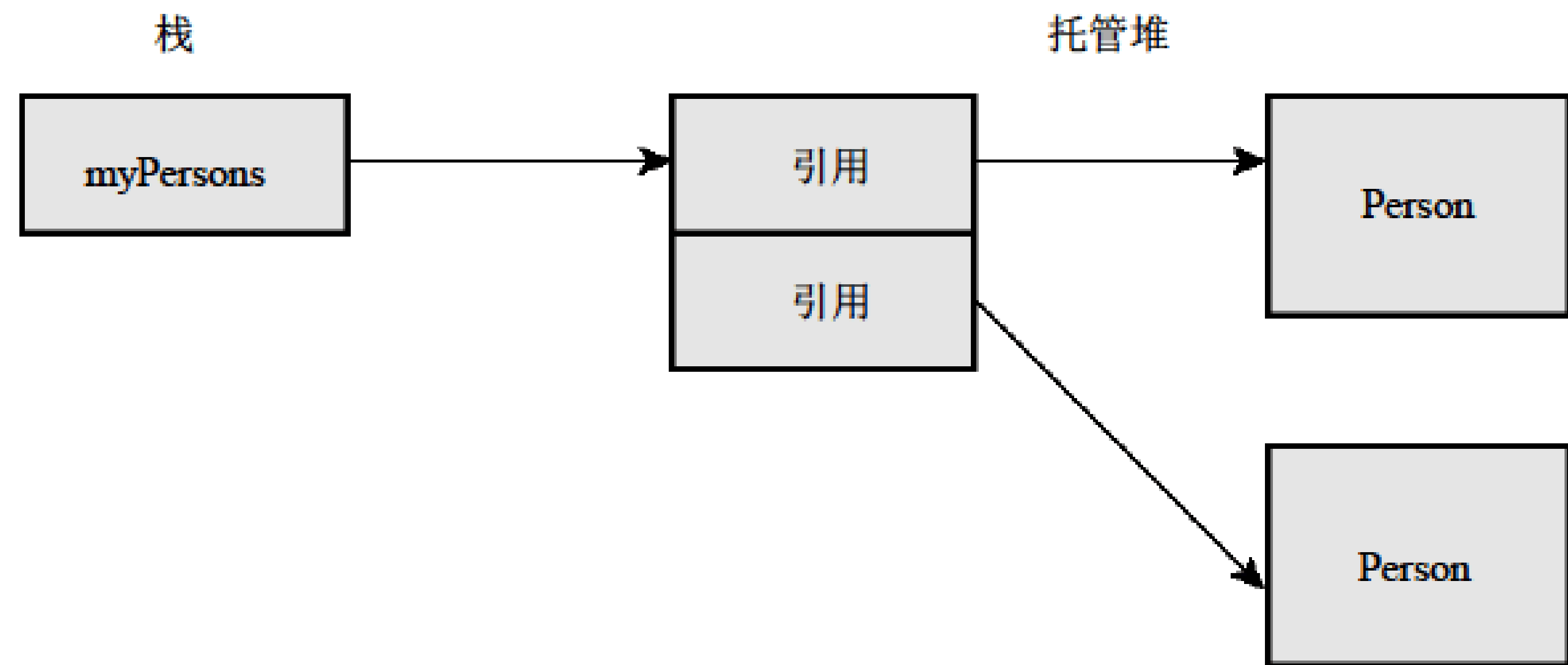


图 7-2

与 int 类型一样，也可以对自定义类型使用数组初始化器：

```
Person[] myPersons2 =
{
  new Person("Ayrton", "Senna"),
  new Person("Michael", "Schumacher")
};
```

## 7.3　多维数组

一般数组(也称为一维数组)用一个整数来索引。多维数组用两个或多个整数来索引。

图 7-3 是二维数组的数学表示法，该数组有 3 行 3 列。第 1 行的值是 1、2 和 3，第 3 行的值是 7、8 和 9。

$$a = \begin{bmatrix} 1, & 2, & 3 \\ 4, & 5, & 6 \\ 7, & 8, & 9 \end{bmatrix}$$

图 7-3

在 C#中声明这个二维数组，需要在方括号中加上一个逗号。数组在初始化时应指定每一维的大小(也称为阶)。接着，就可以使用两个整数作为索引器来访问数组中的元素：

```
int[,] twodim = new int[3, 3];
twodim[0, 0] = 1;
twodim[0, 1] = 2;
twodim[0, 2] = 3;
twodim[1, 0] = 4;
twodim[1, 1] = 5;
twodim[1, 2] = 6;
twodim[2, 0] = 7;
twodim[2, 1] = 8;
twodim[2, 2] = 9;
```

**注意：**
声明数组后，就不能修改其阶数了。

如果事先知道元素的值，就可以使用数组索引器来初始化二维数组。在初始化数组时，使用一个外层的花

括号，每一行用包含在外层花括号中的内层花括号进行初始化。

```
int[,] twodim = {
  {1, 2, 3},
  {4, 5, 6},
  {7, 8, 9}
};
```

> **注意：**
> 使用数组初始化器时，必须初始化数组的每个元素，不能把某些值的初始化放在以后完成。

在花括号中使用两个逗号，就可以声明一个三维数组：

```
int[,,] threedim = {
  { { 1, 2 }, { 3, 4 } },
  { { 5, 6 }, { 7, 8 } },
  { { 9, 10 }, { 11, 12 } }
};
Console.WriteLine(threedim[0, 1, 1]);
```

## 7.4 锯齿数组

二维数组的大小对应于一个矩形，如对应的元素个数为 3×3。而锯齿数组的大小设置比较灵活，在锯齿数组中，每一行都可以有不同的大小。

图 7-4 比较了有 3×3 个元素的二维数组和锯齿数组。图 7-4 中的锯齿数组有 3 行，第 1 行有两个元素，第 2 行有 6 个元素，第 3 行有 3 个元素。

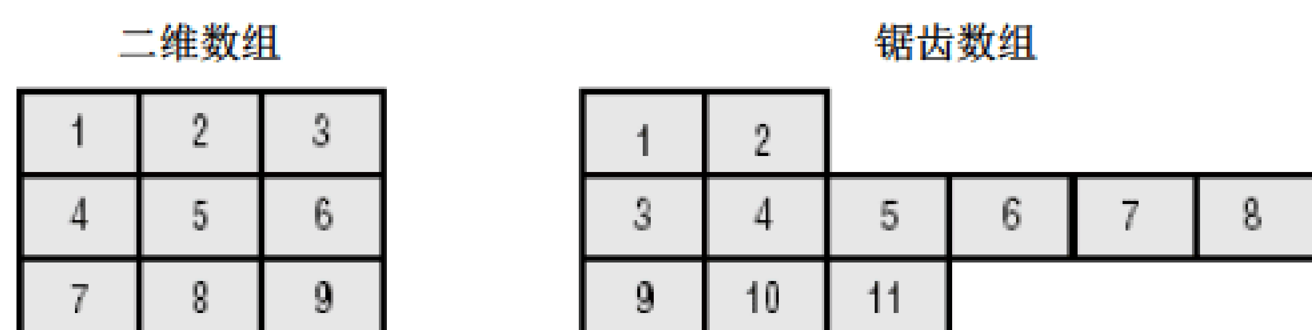



图 7-4

在声明锯齿数组时，要依次放置左右括号。在初始化锯齿数组时，只在第 1 对方括号中设置该数组包含的行数。定义各行中元素个数的第 2 个方括号设置为空，因为这类数组的每一行包含不同的元素个数。之后，为每一行指定行中的元素个数(代码文件 SimpleArrays/Program.cs)：

```
int[][] jagged = new int[3][];
jagged[0] = new int[2] { 1, 2 };
jagged[1] = new int[6] { 3, 4, 5, 6, 7, 8 };
jagged[2] = new int[3] { 9, 10, 11 };
```

迭代锯齿数组中所有元素的代码可以放在嵌套的 for 循环中。在外层的 for 循环中迭代每一行，在内层的 for 循环中迭代一行中的每个元素：

```
for (int row = 0; row < jagged.Length; row++)
{
  for (int element = 0; element < jagged[row].Length; element++)
  {
    Console.WriteLine($"row: {row}, element: {element}, " +
      $"value: {jagged[row][element]}");
  }
}
```

该迭代结果显示了所有的行和每一行中的各个元素：

```
row: 0, element: 0, value: 1
row: 0, element: 1, value: 2
row: 1, element: 0, value: 3
row: 1, element: 1, value: 4
row: 1, element: 2, value: 5
row: 1, element: 3, value: 6
row: 1, element: 4, value: 7
row: 1, element: 5, value: 8
```

```
row: 2, element: 0, value: 9
row: 2, element: 1, value: 10
row: 2, element: 2, value: 11
```

# 7.5 Array 类

用方括号声明数组是 C#中使用 Array 类的表示法。在后台使用 C#语法，会创建一个派生自抽象基类 Array 的新类。这样，就可以使用 Array 类为每个 C#数组定义的方法和属性了。例如，前面就使用了 Length 属性，或者使用 foreach 语句迭代数组。其实这是使用了 Array 类中的 GetEnumerator()方法。

Array 类实现的其他属性有 LongLength 和 Rank。如果数组包含的元素个数超出了整数的取值范围，就可以使用 LongLength 属性来获得元素个数。使用 Rank 属性可以获得数组的维数。

下面通过了解不同的功能来看看 Array 类的其他成员。

## 7.5.1 创建数组

Array 类是一个抽象类，所以不能使用构造函数来创建数组。但除了可以使用 C#语法创建数组实例之外，还可以使用静态方法 CreateInstance()创建数组。如果事先不知道元素的类型，该静态方法就非常有用，因为类型可以作为 Type 对象传递给 CreateInstance()方法。

下面的例子说明了如何创建类型为 int、大小为 5 的数组。CreateInstance()方法的第 1 个参数应是元素的类型，第 2 个参数定义数组的大小。可以用 SetValue()方法设置对应元素的值，用 GetValue()方法读取对应元素的值(代码文件 SimpleArrays/Program.cs)：

```
Array intArray1 = Array.CreateInstance(typeof(int), 5);
for (int i = 0; i < 5; i++)
{
  intArray1.SetValue(33, i);
}

for (int i = 0; i < 5; i++)
{
  Console.WriteLine(intArray1.GetValue(i));
}
```

还可以将已创建的数组强制转换成声明为 int[]的数组：

```
int[] intArray2 = (int[])intArray1;
```

CreateInstance()方法有许多重载版本，可以创建多维数组和不基于 0 的数组。下面的例子创建了一个包含 2×3 个元素的二维数组。第一维基于 1，第二维基于 10：

```
int[] lengths = { 2, 3 };
int[] lowerBounds = { 1, 10 };
Array racers = Array.CreateInstance(typeof(Person), lengths, lowerBounds);
```

SetValue()方法设置数组的元素，其参数是每一维的索引：

```
racers.SetValue(new Person("Alain", "Prost"), 1, 10);
racers.SetValue(new Person("Emerson", "Fittipaldi", 1, 11);
racers.SetValue(new Person("Ayrton", "Senna"), 1, 12);
racers.SetValue(new Person("Michael", "Schumacher"), 2, 10);
racers.SetValue(new Person("Fernando", "Alonso"), 2, 11);
racers.SetValue(new Person("Jenson", "Button"), 2, 12);
```

尽管数组不是基于 0，但可以用一般的 C#表示法为它赋予一个变量。只需要注意不要超出边界即可：

```
Person[,] racers2 = (Person[,])racers;
Person first = racers2[1, 10];
Person last = racers2[2, 12];
```

## 7.5.2 复制数组

因为数组是引用类型，所以将一个数组变量赋予另一个数组变量，就会得到两个引用同一数组的变量。而复制数组，会使数组实现 ICloneable 接口。这个接口定义的 Clone()方法会创建数组的浅表副本。

如果数组的元素是值类型，以下代码段就会复制所有值，如图 7-5 所示：

```
int[] intArray1 = {1, 2};
int[] intArray2 = (int[])intArray1.Clone();
```

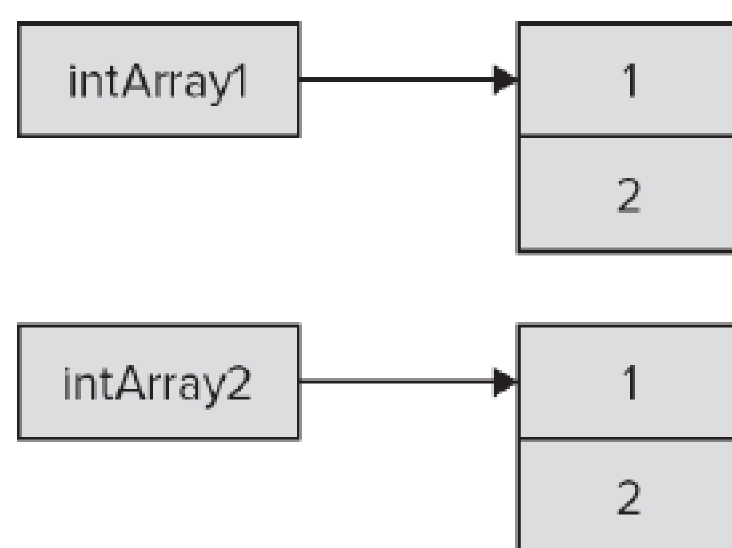



图 7-5

如果数组包含引用类型，则不复制元素，而只复制引用。图 7-6 显示了变量 beatles 和 beatlesClone，其中 beatlesClone 通过从 beatles 中调用 Clone()方法来创建。beatles 和 beatlesClone 引用的 Person 对象是相同的。如果修改 beatlesClone 中一个元素的属性，就会改变 beatles 中的对应对象(代码文件 SimpleArray/Program.cs)。

```
Person[] beatles = {
new Person { FirstName="John", LastName="Lennon" },
new Person { FirstName="Paul", LastName="McCartney" }
};
Person[] beatlesClone = (Person[])beatles.Clone();
```

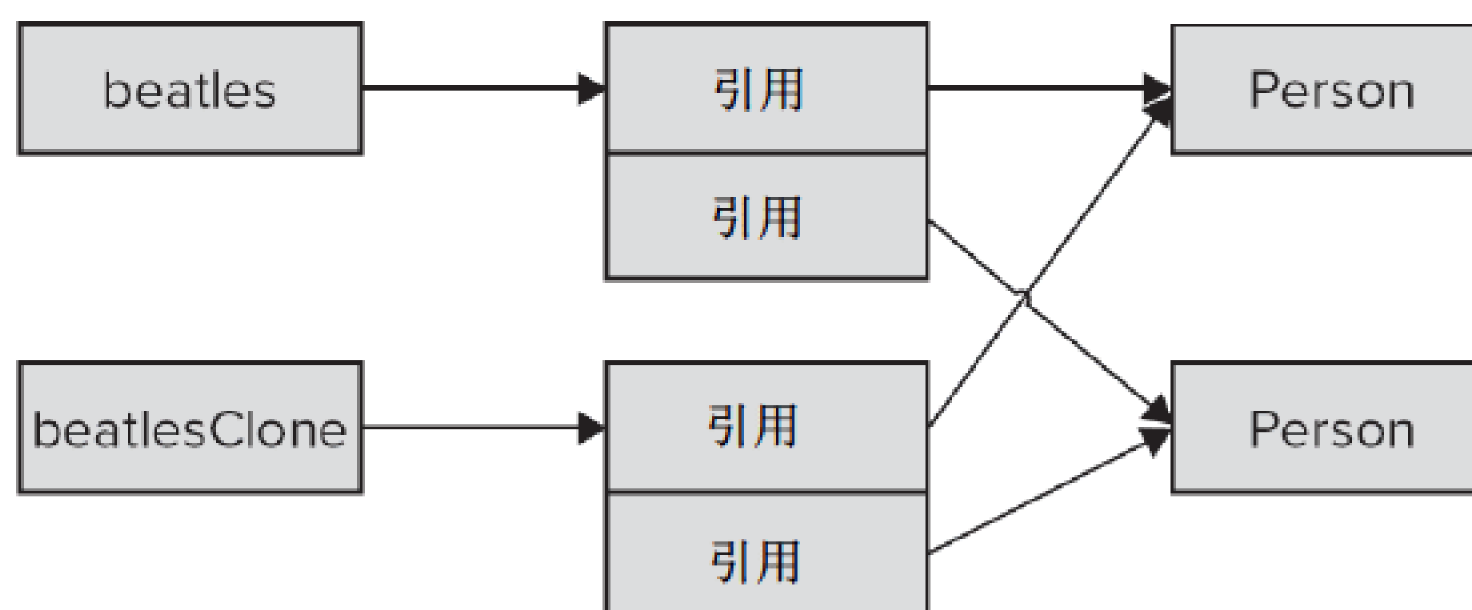



图 7-6

除了使用 Clone()方法之外，还可以使用 Array.Copy()方法创建浅表副本。但 Clone()方法和 Copy()方法有一个重要区别：Clone()方法会创建一个新数组，而 Copy()方法必须传递阶数相同且有足够元素的已有数组。

**注意：**
*如果需要包含引用类型的数组的深层副本，就必须迭代数组并创建新对象。*

### 7.5.3　排序

Array 类使用 Quicksort 算法对数组中的元素进行排序。Sort()方法需要数组中的元素实现 IComparable 接口。因为简单类型(如 System.String 和 System.Int32)实现 IComparable 接口，所以可以对包含这些类型的元素排序。

在示例程序中，数组 name 包含 string 类型的元素，这个数组可以排序(代码文件 SortingSample/Program.cs)。

```
string[] names = {
  "Christina Aguilera",
  "Shakira",
  "Beyonce",
  "Lady Gaga"
};
Array.Sort(names);
foreach (var name in names)
{
  Console.WriteLine(name);
}
```

该应用程序的输出是排好序的数组：

```
Beyonce
Christina Aguilera
Lady Gaga
Shakira
```

如果对数组使用自定义类，就必须实现 IComparable 接口。这个接口只定义了一个方法 CompareTo()，如果要比较的对象相等，该方法就返回 0。如果该实例应排在参数对象的前面，该方法就返回小于 0 的值。如果该实例应排在参数对象的后面，该方法就返回大于 0 的值。

修改 Person 类，使之实现 IComparable<Person>接口。先使用 String 类中的 CompareTo()方法对 LastName 的值进行比较。如果 LastName 的值相同，就比较 FirstName(代码文件 SortingSample/Person.cs)：

```
public class Person: IComparable<Person>
{
  public int CompareTo(Person other)
  {
    if (other == null) return 1;
    int result = string.Compare(this.LastName, other.LastName);
    if (result == 0)
    {
      result = string.Compare(this.FirstName, other.FirstName);
    }
    return result;
  }
//...
```

现在可以按照姓氏对 Person 对象对应的数组排序(代码文件 SortingSample/Program.cs)：

```
Person[] persons = {
  new Person("Damon", "Hill"),
  new Person("Niki", "Lauda"),
  new Person("Ayrton", "Senna"),
  new Person("Graham", "Hill")
};
Array.Sort(persons);
foreach (var p in persons)
{
  Console.WriteLine(p);
}
```

使用 Person 类的排序功能，会得到按姓氏排序的姓名：

```
Damon Hill
Graham Hill
Niki Lauda
Ayrton Senna
```

如果 Person 对象的排序方式与上述不同，或者不能修改在数组中用作元素的类，就可以实现 IComparer 接口或 IComparer<T>接口。这两个接口定义了方法 Compare()。要比较的类必须实现这两个接口之一。IComparer 接口独立于要比较的类。这就是 Compare()方法定义了两个要比较的参数的原因。其返回值与 IComparable 接口的 CompareTo()方法类似。

类 PersonComparer 实现了 IComparer<Person>接口，可以按照 firstName 或 lastName 对 Person 对象排序。枚举 PersonCompareType 定义了可用于 PersonComparer 的排序选项：FirstName 和 LastName。排序方式由 PersonComparer 类的构造函数定义，在该构造函数中设置了一个 PersonCompareType 值。实现 Compare()方法时用一个 switch 语句指定是按 FirstName 还是 LastName 排序(代码文件 SortingSample/PersonComparer.cs)。

```
public enum PersonCompareType
{
  FirstName,
  LastName
}

public class PersonComparer: IComparer<Person>
{
  private PersonCompareType _compareType;
  public PersonComparer(PersonCompareType compareType) =>
    _compareType = compareType;

  public int Compare(Person x, Person y)
```

```
    {
      if (x is null && y is null) return 0;
      if (x is null) return 1;
      if (y is null) return -1;
      switch (_compareType)
      {
        case PersonCompareType.FirstName:
          return string.Compare(x.FirstName, y.FirstName);
        case PersonCompareType.LastName:
          return string.Compare(x.LastName, y.LastName);
        default:
          throw new ArgumentException("unexpected compare type");
      }
    }
  }
```

现在，可以将一个 PersonComparer 对象传递给 Array.Sort()方法的第 2 个参数。下面按名字对 persons 数组排序(代码文件 SortingSample/Program.cs)：

```
Array.Sort(persons, new PersonComparer(PersonCompareType.FirstName));
foreach (var p in persons)
{
  Console.WriteLine(p);
}
```

persons 数组现在按名字排序：

```
Ayrton Senna
Damon Hill
Graham Hill
Niki Lauda
```

**注意：**

Array 类还提供了 Sort 方法，它需要将一个委托作为参数。这个参数可以传递给方法，从而比较两个对象，而不需要依赖 IComparable 或 IComparer 接口。第 8 章将介绍如何使用委托。

## 7.6　数组作为参数

数组可以作为参数传递给方法，也可以从方法返回。要返回一个数组，只需要把数组声明为返回类型，如下面的方法 GetPersons()所示：

```
static Person[] GetPersons() =>
  new Person[] {
    new Person("Damon", "Hill"),
    new Person("Niki", "Lauda"),
    new Person("Ayrton", "Senna"),
    new Person("Graham", "Hill")
  };
```

要把数组传递给方法，应把数组声明为参数，如下面的 DisplayPersons()方法所示：

```
static void DisplayPersons(Person[] persons)
{
  //...
}
```

## 7.7　数组协变

数组支持协变。这表示数组可以声明为基类，其派生类型的元素可以赋予数组元素。

例如，可以声明一个 object[]类型的参数，给它传递一个 Person[]：

```
static void DisplayArray(object[] data)
{
  //...
}
```

**注意：**

数组协变只能用于引用类型，不能用于值类型。另外，数组协变有一个问题，它只能通过运行时异常来解决。如果把 Person 数组赋予 object 数组，object 数组就可以使用派生自 object 的任何元素。例如，编译器允许把字符串传递给数组元素。但因为 object 数组引用 Person 数组，所以会出现一个运行时异常 ArrayTypeMismatchException。

## 7.8 枚举

在 foreach 语句中使用枚举，可以迭代集合中的元素，且不需要知道集合中的元素个数。foreach 语句使用了一个枚举器。图 7-7 显示了调用 foreach 方法的客户端和集合之间的关系。数组或集合实现带 GetEumerator()方法的 IEumerable 接口。GetEumerator()方法返回一个实现 IEumerator 接口的枚举。接着，foreach 语句就可以使用 IEnumerator 接口迭代集合了。

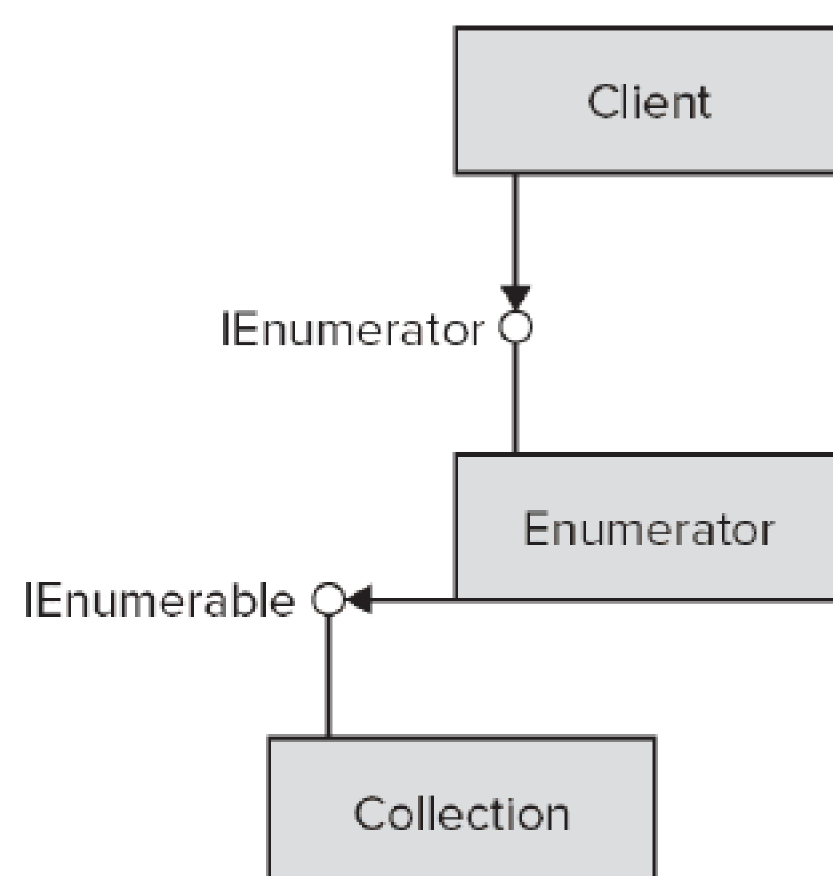



图 7-7

**注意：**

GetEnumerator()方法用 IEnumerable 接口定义。foreach 语句并不需要在集合类中实现这个接口。有一个名为 GetEnumerator()的方法，它返回实现了 IEnumerator 接口的对象就足够了。

### 7.8.1 IEnumerator 接口

foreach 语句使用 IEnumerator 接口的方法和属性，迭代集合中的所有元素。为此，IEnumerator 定义了 Current 属性，来返回光标所在的元素，该接口的 MoveNext()方法移动到集合的下一个元素上。如果有这个元素，该方法就返回 true。如果集合不再有更多的元素，该方法就返回 false。

这个接口的泛型版本 IEnumerator<T>派生自接口 IDisposable，因此定义了 Dispose()方法，来清理给枚举器分配的资源。

**注意：**

IEnumerator 接口还定义了 Reset()方法，以与 COM 交互操作。许多.NET 枚举器通过抛出 NotSupportedException 类型的异常，来实现这个方法。

### 7.8.2 foreach 语句

C#的 foreach 语句不会解析为 IL 代码中的 foreach 语句。C#编译器会把 foreach 语句转换为 IEnumerator 接口的方法和属性。下面是一条简单的 foreach 语句，它迭代 persons 数组中的所有元素，并逐个显示它们：

```
foreach (var p in persons)
{
```

```
  Console.WriteLine(p);
}
```

foreach 语句会解析为下面的代码段。首先，调用 GetEnumerator()方法，获得数组的一个枚举器。在 while 循环中——只要 MoveNext()返回 true——就用 Current 属性访问数组中的元素：

```
IEnumerator<Person> enumerator = persons.GetEnumerator();
while (enumerator.MoveNext())
{
  Person p = enumerator.Current;
  Console.WriteLine(p);
}
```

### 7.8.3　yield 语句

自 C#的第 1 个版本以来，使用 foreach 语句可以轻松地迭代集合。在 C# 1.0 中，创建枚举器仍需要做大量的工作。C# 2.0 添加了 yield 语句，以便于创建枚举器。yield return 语句返回集合的一个元素，并移动到下一个元素上。yield break 可停止迭代。

下一个例子是用 yield return 语句实现一个简单集合的代码。HelloCollection 类包含 GetEnumerator()方法。该方法的实现代码包含两条 yield return 语句，它们分别返回字符串 Hello 和 World(代码文件 YieldSample/Program.cs)。

```
using System;
using System.Collections;
namespace Wrox.ProCSharp.Arrays
{
  public class HelloCollection
  {
    public IEnumerator<string> GetEnumerator()
    {
      yield return "Hello";
      yield return "World";
    }
  }
}
```

**注意：**

*包含 yield 语句的方法或属性也称为迭代块。迭代块必须声明为返回 IEnumerator 或 IEnumerable 接口，或者这些接口的泛型版本。这个块可以包含多条 yield return 语句或 yield break 语句，但不能包含 return 语句。*

现在可以用 foreach 语句迭代集合了：

```
public void HelloWorld()
{
  var helloCollection = new HelloCollection();
  foreach (var s in helloCollection)
  {
    Console.WriteLine(s);
  }
}
```

使用迭代块，编译器会生成一个 yield 类型，其中包含一个状态机，如下面的代码段所示。yield 类型实现 IEnumerator 和 IDisposable 接口的属性和方法。在下面的例子中，可以把 yield 类型看作内部类 Enumerator。外部类的 GetEnumerator()方法实例化并返回一个新的 yield 类型。在 yield 类型中，变量 state 定义了迭代的当前位置，每次调用 MoveNext()时，当前位置都会改变。MoveNext()封装了迭代块的代码，并设置了 current 变量的值，从而使 Current 属性根据位置返回一个对象。

```
public class HelloCollection
{
  public IEnumerator GetEnumerator() => new Enumerator(0);

  public class Enumerator: IEnumerator<string>, IEnumerator, IDisposable
  {
    private int _state;
    private string _current;
```

```
    public Enumerator(int state) => _state = state;
    bool System.Collections.IEnumerator.MoveNext()
    {
      switch (state)
      {
        case 0:
          _current = "Hello";
          _state = 1;
          return true;
        case 1:
          _current = "World";
          _state = 2;
          return true;
        case 2:
          break;
      }
      return false;
    }

    void System.Collections.IEnumerator.Reset() =>
      throw new NotSupportedException();

    string System.Collections.Generic.IEnumerator<string>.Current => current;
    object System.Collections.IEnumerator.Current => current;
    void IDisposable.Dispose() { }
  }
}
```

**注意：**

yield 语句会生成一个枚举器，而不仅仅生成一个包含的项的列表。这个枚举器通过 foreach 语句调用。从 foreach 中依次访问每一项时，就会访问枚举器。这样就可以迭代大量的数据，而不需要一次把所有的数据都读入内存。

### 1. 迭代集合的不同方式

在下面这个比 Hello World 示例略大但比较真实的示例中，可以使用 yield return 语句，以不同方式迭代集合的类。类 MusicTitles 可以用默认方式通过 GetEnumerator()方法迭代标题，用 Reverse()方法逆序迭代标题，用 Subset()方法迭代子集(代码文件 YieldSample/MusicTitles.cs)：

```
public class MusicTitles
{
  string[] names = {"Tubular Bells", "Hergest Ridge", "Ommadawn", "Platinum"};

  public IEnumerator<string> GetEnumerator()
  {
    for (int i = 0; i < 4; i++)
    {
      yield return names[i];
    }
  }

  public IEnumerable<string> Reverse()
  {
    for (int i = 3; i >= 0; i-)
    {
      yield return names[i];
    }
  }

  public IEnumerable<string> Subset(int index, int length)
  {
    for (int i = index; i < index + length; i++)
    {
      yield return names[i];
    }
  }
}
```

**注意：**

类支持的默认迭代是定义为返回 IEnumerator 的 GetEnumerator()方法。命名的迭代返回 IEnumerable。

迭代字符串数组的客户端代码先使用 GetEnumerator()方法，该方法不必在代码中编写，因为这是 foreach 语句默认使用的方法。然后逆序迭代标题，最后将索引和要迭代的项数传递给 Subset()方法，来迭代子集(代码文件 YieldSample/Program.cs)：

```
var titles = new MusicTitles();
foreach (var title in titles)
{
  Console.WriteLine(title);
}
Console.WriteLine();

Console.WriteLine("reverse");
foreach (var title in titles.Reverse())
{
  Console.WriteLine(title);
}
Console.WriteLine();

Console.WriteLine("subset");
foreach (var title in titles.Subset(2, 2))
{
  Console.WriteLine(title);
}
```

### 2. 用 yield return 返回枚举器

使用 yield 语句还可以完成更复杂的任务，例如，从 yield return 中返回枚举器。在 Tic-Tac-Toe 游戏中有 9 个域，玩家轮流在这些域中放置一个“十”字或一个圆。这些移动操作由 GameMoves 类模拟。方法 Cross()和 Circle()是创建迭代类型的迭代块。变量 cross 和 circle 在 GameMoves 类的构造函数中设置为 Cross()和 Circle()方法。这些字段不设置为调用的方法，而是设置为用迭代块定义的迭代类型。在 Cross()迭代块中，将移动操作的信息写到控制台上，并递增移动次数。如果移动次数大于 8，就用 yield break 停止迭代；否则，就在每次迭代中返回 yield 类型 circle 的枚举对象。Circle()迭代块非常类似于 Cross()迭代块，只是它在每次迭代中返回 cross 迭代器类型(代码文件 YieldSample/GameMoves.cs)。

```
public class GameMoves
{
  private IEnumerator _cross;
  private IEnumerator _circle;
  public GameMoves()
  {
    _cross = Cross();
    _circle = Circle();
  }
  private int _move = 0;
  const int MaxMoves = 9;

  public IEnumerator Cross()
  {
    while (true)
    {
      Console.WriteLine($"Cross, move {_move}");
      if (++_move >= MaxMoves)
      {
        yield break;
      }
      yield return _circle;
    }
  }

  public IEnumerator Circle()
  {
    while (true)
    {
      Console.WriteLine($"Circle, move {move}");
      if (++_move >= MaxMoves)
      {
        yield break;
```

```
        }
        yield return _cross;
      }
    }
  }
```

在客户端程序中，可以以如下方式使用 GameMoves 类。将枚举器设置为由 game.Cross()返回的枚举器类型，以设置第一次移动。在 while 循环中，调用 enumerator.MoveNext()。第一次调用 enumerator.MoveNext()时，会调用 Cross()方法，Cross()方法使用 yield 语句返回另一个枚举器。返回的值可以用 Current 属性访问，并设置为 enumerator 变量，用于下一次循环：

```
var game = new GameMoves();
IEnumerator enumerator = game.Cross();
while (enumerator.MoveNext())
{
  enumerator = enumerator.Current as IEnumerator;
}
```

这个程序的输出会显示交替移动的情况，直到最后一次移动：

```
Cross, move 0
Circle, move 1
Cross, move 2
Circle, move 3
Cross, move 4
Circle, move 5
Cross, move 6
Circle, move 7
Cross, move 8
```

## 7.9 结构比较

数组和元组都实现接口 IStructuralEquatable 和 IStructuralComparable。这两个接口不仅可以比较引用，还可以比较内容。这些接口都是显式实现的，所以在使用时需要把数组和元组强制转换为这个接口。IStructuralEquatable 接口用于比较两个元组或数组是否有相同的内容，IStructuralComparable 接口用于给元组或数组排序。

**注意：**
元组参见第 13 章。

对于说明 IStructuralEquatable 接口的示例，使用实现 IEquatable 接口的 Person 类。IEquatable 接口定义了一个强类型化的 Equals()方法，以比较 FirstName 和 LastName 属性的值(代码文件 StructuralComparison/Person.cs)：

```
public class Person: IEquatable<Person>
{
  public int Id { get; }
  public string FirstName { get; }
  public string LastName { get; }

  public Person(int id, string firstName, string lastName)
  {
    Id = id;
    FirstName = firstName;
    LastName = lastName;
  }

  public override string ToString() => $"{Id}, {FirstName} {LastName}";

  public override bool Equals(object obj)
  {
    if (obj == null)
    {
      return base.Equals(obj);
    }
    return Equals(obj as Person);
  }
```

```
  public override int GetHashCode() => Id.GetHashCode();

  public bool Equals(Person other)
  {
    if (other == null)
    return base.Equals(other);
    return Id == other.Id && FirstName == other.FirstName &&
    LastName == other.LastName;
  }
}
```

现在创建了两个包含 Person 项的数组。这两个数组通过变量名 janet 包含相同的 Person 对象，和两个内容相同的不同 Person 对象。比较运算符“!=”返回 true，因为这其实是两个变量 persons1 和 persons2 引用的两个不同数组。因为 Array 类没有重写带一个参数的 Equals()方法，所以用“==”运算符比较引用也会得到相同的结果，即这两个变量不相同(代码文件 StructuralComparison/Program.cs)：

```
var janet = new Person("Janet", "Jackson");
Person[] people1 = {
  new Person("Michael", "Jackson"),
  janet
};
Person[] people2 = {
  new Person("Michael", "Jackson")
  janet
};

if (people1 != people2)
{
  Console.WriteLine("not the same reference");
}
```

对于 IStructuralEquatable 接口定义的 Equals()方法，它的第一个参数是 object 类型，第二个参数是 IEqualityComparer 类型。调用这个方法时，通过传递一个实现了 IEqualityComparer<T>的对象，就可以定义如何进行比较。通过 EqualityComparer<T>类完成 IEqualityComparer 的一个默认实现。这个实现检查该类型是否实现了 IEquatable 接口，并调用 IEquatable.Equals()方法。如果该类型没有实现 IEquatable，就调用 Object 基类中的 Equals()方法进行比较。

Person 实现 IEquatable<Person>，在此过程中比较对象的内容，而数组的确包含相同的内容：

```
if ((people1 as IStructuralEquatable).Equals(people2,
  EqualityComparer<Person>.Default))
{
  Console.WriteLine("the same content");
}
```

## 7.10　Span

为了快速访问托管或非托管的连续内存，可以使用 Span<T>结构。一个可以使用 Span<T>的例子是数组；Span<T> 结构在后台保存在连续的内存中。另一个例子是长字符串。在第 9 章中使用了 Span<T>和字符串。

使用 Span<T>，可以直接访问数组元素。数组的元素没有复制，但是它们可以直接使用，这比复制要快。

在下面的代码片段中，首先创建并初始化一个简单的 int 数组。调用构造函数，并将数组传递给 Span<int>，以创建一个 Span<int>对象。Span<T>类型提供了一个索引器，因此可以使用这个索引器访问 Span<T>的元素。这里，第二个元素的值改为 11。由于数组 arr1 是在 span 中引用的，因此通过改变 span <T>元素来改变数组的第二个元素(代码文件 SpanSample/Program.cs)：

```
private static Span<int> IntroSpans()
{
  int[] arr1 = { 1, 4, 5, 11, 13, 18 };
  var span1 = new Span<int>(arr1);
  span1[1] = 11;
  Console.WriteLine($"arr1[1] is changed via span1[1]: {arr1[1]}");
  return span1;
}
```

### 7.10.1 创建切片

Span<T>的一个强大特性是，可以使用它访问数组的部分或切片。使用切片时，不会复制数组元素，它们是从 Span 中直接访问的。

下面的代码片段展示了创建切片的两种方法。第一种方法是，使用一个构造函数重载版本传递应使用的数组的开头和长度。使用变量 span3 引用这个新创建的 Span<int>，它只能访问数组 arr2 的 3 个元素，从第四个元素开始。构造函数还有另一个重载版本，它可以仅传递切片的开头。在这个重载版本中，会提取数组的剩余部分，一直到数组的末尾。调用 Slice 方法，也可以从 Span<T>对象中创建一个切片。它有类似的重载版本。通过变量 span4，使用之前创建的 span1 创建一个包含 4 个元素的切片，且从 span1 的第 3 个元素开始(代码文件 SpanSample/Program.cs)：

```
private static Span<int> CreateSlices(Span<int> span1)
{
  Console.WriteLine(nameof(CreateSlices));
  int[] arr2 = { 3, 5, 7, 9, 11, 13, 15 };
  var span2 = new Span<int>(arr2);
  var span3 = new Span<int>(arr2, start: 3, length: 3);
  var span4 = span1.Slice(start: 2, length: 4);

  DisplaySpan("content of span3", span3);
  DisplaySpan("content of span4", span4);
  Console.WriteLine();
  return span2;
}
```

DisplaySpan 方法用于显示 span 的内容。下面代码段中的方法利用了 ReadOnlySpan。如果不需要更改 span 引用的内容，就可以使用这个 span 类型，DisplaySpan 方法就是这样。ReadOnlySpan<T>在本章后面详细讨论：

```
private static void DisplaySpan(string title, ReadOnlySpan<int> span)
{
  Console.WriteLine(title);
  for (int i = 0; i < span.Length; i++)
  {
    Console.Write($"{span[i]}.");
  }
  Console.WriteLine();
}
```

运行应用程序时，显示 span3 和 span4 的内容，它们是 arr2 和 arr1 的子集。

```
content of span3
9.11.13.
content of span4
6.8.10.12.
```

**注意：**

Span<T>是安全的，不会出界。如果在创建的 span 超出包含的数组长度，就会抛出 ArgumentOutOfRangeException 类型的异常。阅读第 14 章，了解关于异常处理的更多信息。

### 7.10.2 使用 Span 改变值

前面介绍了如何使用 Span <T>类型的索引器，直接更改由 span 引用的数组元素。下面的代码片段显示了更多的选项。

可以调用 Clear 方法，该方法用 0 填充包含 int 类型的 span；可以调用 Fill 方法，用传递给 Fill 方法的值来填充 span；可以将一个 Span<T>复制到另一个 Span<T>。在 CopyTo 方法中，如果目标 span 不够大，就会抛出 ArgumentException 类型的异常。可以使用 TryCopyTo 方法来避免这个结果。如果目标 span 不够大，此方法不会抛出异常；而是返回 false，因为复制没有成功 (代码文件 SpanSample/Program.cs)：

```
private static void ChangeValues(Span<int> span1, Span<int> span2)
{
  Console.WriteLine(nameof(ChangeValues));
  Span<int> span4 = span1.Slice(start: 4);
```

```
    span4.Clear();
    DisplaySpan("content of span1", span1);
    Span<int> span5 = span2.Slice(start: 3, length: 3);
    span5.Fill(42);
    DisplaySpan("content of span2", span2);
    span5.CopyTo(span1);
    DisplaySpan("content of span1", span1);

    if (!span1.TryCopyTo(span4))
    {
      Console.WriteLine("Couldn't copy span1 to span4 because span4 is " +
        "too small");
      Console.WriteLine($"length of span4: {span4.Length}, length of " +
        $"span1: {span1.Length}");
    }
    Console.WriteLine();
  }
```

运行应用程序时，可以看到 span1 的内容，其中的最后两个数使用 span4 清除，还可以看到 span2 的内容，其中有三个元素用 span5 来填充值 42，也可以看到 span1 的内容，其中前三个数字从 span5 中复制。从 span1 复制到 span4 是不成功的，因为 span4 的长度只有 4，而 span1 的长度是 6：

```
content of span1
2.11.6.8.0.0.
content of span2
3.5.7.42.42.42.15.
content of span1
42.42.42.8.0.0.
Couldn't copy span1 to span4 because span4 is too small
length of span4: 2, length of span1: 6
```

### 7.10.3　只读的 Span

如果只需要对数组段进行读访问，就可以使用 ReadOnlySpan<T>，如前面的 DisplaySpan 方法所示。对于 ReadOnlySpan<T>，索引器是只读的，这种类型没有提供 Clear 和 Fill 方法。但是，可以调用 CopyTo 方法，将 ReadOnlySpan<T>的内容复制到 Span<T>。

下面的代码片段使用 ReadOnlySpan<T>的构造函数从一个数组中创建了 readOnlySpan1，readOnlySpan2 和 readOnlySpan3 是由 Span<int>和 int[]的直接赋予创建的。隐式转换操作符可用于 ReadOnlySpan<T>(代码文件 SpanSample/Program.cs)：

```
private static void ReadonlySpan(Span<int> span1)
{
    Console.WriteLine(nameof(ReadonlySpan));
    int[] arr = span1.ToArray();
    ReadOnlySpan<int> readOnlySpan1 = new ReadOnlySpan<int>(arr);
    DisplaySpan("readOnlySpan1", readOnlySpan1);

    ReadOnlySpan<int> readOnlySpan2 = span1;
    DisplaySpan("readOnlySpan2", readOnlySpan2);
    ReadOnlySpan<int> readOnlySpan3 = arr;
    DisplaySpan("readOnlySpan3", readOnlySpan3);
    Console.WriteLine();
}
```

**注意：**

本书的先前版本演示了 ArraySegment<T>的用法，尽管 ArraySegment<T>仍然可用，但它有一些缺点，可以使用更灵活的 Span<T>作为替换。如果已经使用了 ArraySegment<T>，可以保留代码并与 span 交互。Span<T>的构造函数也允许传递 ArraySegment<T>来创建 Span<T>实例。

## 7.11　数组池

如果一个应用程序创建和销毁了许多数组，垃圾收集器就有一些工作要做。为了减少垃圾收集器的工作，可以通过 ArrayPool 类使用数组池。ArrayPool 管理一个数组池。数组可以从这里租借，并返回到池中。内存在

ArrayPool 中管理。

### 7.11.1 创建数组池

通过调用静态 Create 方法，可以创建 ArrayPool<T>。为了提高效率，数组池在多个桶中为大小类似的数组管理内存。使用 Create 方法，可以在需要一个桶之前，在另一个桶中定义最大的数组长度和数组的数量：

```
ArrayPool<int> customPool = ArrayPool<int>.Create(
  maxArrayLength: 40000, maxArraysPerBucket: 10);
```

maxArrayLength 的默认值是 1024 * 1024 字节，maxArraysPerBucket 的默认值是 50。数组池使用多个桶，以便在使用多个数组时更快地访问数组。只要还没有到达数组的最大数量，大小类似的数组就尽可能保存在同一个桶中。

还可以通过访问 ArrayPool<T>类的共享属性，来使用预定义的共享池：

```
ArrayPool<int> sharedPool = ArrayPool<int>.Shared;
```

### 7.11.2 从池中租用内存

调用 Rent 方法可以请求池中的内存。Rent 方法接受应请求的最小数组长度。如果池中已经有内存，则返回该内存。如果它不可用，就给池分配内存，然后返回。在下面的代码片段中，在 for 循环中请求一个包含 1024、2048、3096 等元素的数组(代码文件 ArrayPoolSample/Program.cs)：

```
private static void UseSharedPool()
{
  for (int i = 0; i < 10; i++)
  {
    int arrayLength = (i + 1) << 10;
    int[] arr = ArrayPool<int>.Shared.Rent(arrayLength);
    Console.WriteLine($"requested an array of {arrayLength} " +
      $"and received {arr.Length}");
    //...
  }
}
```

Rent 方法返回一个数组，其中至少包含所请求的元素个数。返回的数组可能有更多的可用内存。共享池中至少有 16 个元素。托管数组的元素计数总是重复的——例如，16、32、64、128、256、512、1024、2048、4096、8192 个元素等。

运行应用程序时，如果请求的数组大小不符合池管理的数组，就返回较大的数组：

```
requested an array of 1024 and received 1024
requested an array of 2048 and received 2048
requested an array of 3072 and received 4096
requested an array of 4096 and received 4096
requested an array of 5120 and received 8192
requested an array of 6144 and received 8192
requested an array of 7168 and received 8192
requested an array of 8192 and received 8192
requested an array of 9216 and received 16384
requested an array of 10240 and received 16384
```

### 7.11.3 将内存返回给池

不再需要数组时，可以将其返回到池中。数组返回后，可以稍后再用另一个 Rent 来重用它。

调用数组池的 Return 方法并将数组传递给 Return 方法，将数组返回到池中。使用一个可选参数，可以指定在返回池之前是否清除该数组。如果不清除它，下一个从池中租用数组的人可以读取数据。清除数据，可以避免这一点，但是需要更多的 CPU 时间(代码文件 ArrayPoolSample/Program.cs)：

```
ArrayPool<int>.Shared.Return(arr, clearArray: true);
```

**注意：**
第 17 章讨论了垃圾收集器以及如何获取内存地址的信息。

## 7.12　小结

本章介绍了创建和使用简单数组、多维数组和锯齿数组的 C#表示法。C#数组在后台使用 Array 类，这样就可以用数组变量调用这个类的属性和方法。

我们还探讨了如何使用 IComparable 和 IComparer 接口给数组中的元素排序，描述了如何创建和使用枚举器、IEnumerable 和 IEnumerator 接口，以及 yield 语句。

最后介绍了如何通过 Span<T> 和 ArrayPool 高效地使用数组。

第 8 章介绍 C#的更多重要功能：委托、lambda 和事件。

第 8 章

# 委托、lambda 表达式和事件

**本章要点**

- 委托
- lambda 表达式
- 闭包
- 事件

**本章源代码下载地址(wrox.com)：**

打开 www.wrox.com 的 Download Code 选项卡可下载本章源代码。源代码也可以在 Delegates 目录的 https://github.com/ProfessionalCSharp/ProfessionalCSharp7 中找到。本章代码分为以下几个主要的示例文件：

- 简单委托(Simple Delegates)
- 冒泡排序(Bubble Sorter)
- lambda 表达式(lambda Expressions)
- 事件示例(Events Sample)

## 8.1 引用方法

委托是寻址方法的.NET 版本。在 C++中，函数指针只不过是一个指向内存位置的指针，它不是类型安全的。我们无法判断这个指针实际指向什么，参数和返回类型等项就更无从知晓了。而.NET 委托完全不同；委托是类型安全的类，它定义了返回类型和参数的类型。委托类不仅包含对方法的引用，也可以包含对多个方法的引用。

lambda 表达式与委托直接相关。当参数是委托类型时，就可以使用 lambda 表达式实现委托引用的方法。

本章介绍委托和 lambda 表达式的基础知识，说明如何通过 lambda 表达式实现委托方法调用，并阐述.NET 如何将委托用作实现事件的方式。

## 8.2 委托

当要把方法传送给其他方法时，就需要使用委托。要了解具体的含义，可以看看下面一行代码：

```
int i = int.Parse("99");
```

我们习惯于把数据作为参数传递给方法，如上面的例子所示。所以，给方法传递另一个方法听起来有点奇怪。而有时某个方法执行的操作并不是针对数据进行的，而是要对另一个方法进行调用。更麻烦的是，在编译时我们不知道第二个方法是什么，这个信息只能在运行时得到，所以需要把第二个方法作为参数传递给第一个方法。这听起来很令人迷惑，下面用几个示例来说明：

- 启动线程和任务——在 C#中，可以告诉计算机并行运行某些新的执行序列，同时运行当前的任务。这种序列就称为线程，在一个基类 System.Threading.Thread 的实例上使用方法 Start()，就可以启动一个线程。如果要告诉计算机启动一个新的执行序列，就必须说明要在哪里启动该序列；必须为计算机提供开始启动的方法的细节，即 Thread 类的构造函数必须带有一个参数，该参数定义了线程调用的方法。
- 通用库类——许多库包含执行各种标准任务的代码。这些库通常可以自我包含，这样在编写库时，就会知道任务该如何执行。但是有时在任务中还包含子任务，只有使用该库的客户端代码才知道如何执行这些子任务。例如，假设要编写一个类，它带有一个对象数组，并把它们按升序排列。但是，排序的部分过程会涉及重复使用数组中的两个对象，比较它们，看看哪一个应放在前面。如果要编写的类必须能对任何对象数组排序，就无法提前告诉计算机应如何比较对象。处理类中对象数组的客户端代码也必须告诉类如何比较要排序的特定对象。换言之，客户端代码必须给类传递某个可以调用并进行这种比较的合适方法的细节。
- 事件——一般的思路是通知代码发生了什么事件。GUI 编程主要处理事件。在引发事件时，运行库需要知道应执行哪个方法。这就需要把处理事件的方法作为一个参数传递给委托。这些将在本章后面讨论。

在 C 和 C++中，只能提取函数的地址，并作为一个参数传递它。C 没有类型安全性，可以把任何函数传递给需要函数指针的方法。但是，这种直接方法不仅会导致一些关于类型安全性的问题，而且没有意识到：在进行面向对象编程时，几乎没有方法是孤立存在的，而是在调用方法前通常需要与类实例相关联。所以.NET Framework 在语法上不允许使用这种直接方法。如果要传递方法，就必须把方法的细节封装在一种新的对象类型中，即委托。委托只是一种特殊类型的对象，其特殊之处在于，我们以前定义的所有对象都包含数据，而委托包含的只是一个或多个方法的地址。

## 8.2.1　声明委托

在 C#中使用一个类时，分两个阶段操作。首先，需要定义这个类，即告诉编译器这个类由什么字段和方法组成。然后(除非只使用静态方法)，实例化该类的一个对象。使用委托时，也需要经过这两个步骤。首先必须定义要使用的委托，对于委托，定义它就是告诉编译器这种类型的委托表示哪种类型的方法。然后，必须创建该委托的一个或多个实例。编译器在后台将创建表示该委托的一个类。

声明委托的语法如下：

```
delegate void IntMethodInvoker(int x);
```

在这个示例中，声明了一个委托 IntMethodInvoker，并指定该委托的每个实例都可以包含一个方法的引用，该方法带有一个 int 参数，并返回 void。理解委托的一个要点是它们的类型安全性非常高。在定义委托时，必须给出它所表示的方法的签名和返回类型等全部细节。

**注意：**
*理解委托的一种好方式是把委托视为给方法的签名和返回类型指定名称。*

假定要定义一个委托 TwoLongsOp，该委托表示的方法有两个 long 型参数，返回类型为 double。可以编写如下代码：

```
delegate double TwoLongsOp(long first, long second);
```

或者要定义一个委托，它表示的方法不带参数，返回一个 string 型的值，可以编写如下代码：

```
delegate string GetAString();
```

其语法类似于方法的定义，但没有方法主体，且定义的前面要加上关键字 delegate。因为定义委托基本上是定义一个新类，所以可以在定义类的任何相同地方定义委托。也就是说，可以在另一个类的内部定义委托，也可以在任何类的外部定义，还可以在名称空间中把委托定义为顶层对象。根据定义的可见性和委托的作用域，可以在委托的定义上应用任意常见的访问修饰符：public、private、protected 等：

```
public delegate string GetAString();
```

**注意：**

实际上，"定义一个委托"是指"定义一个新类"。委托实现为派生自基类 System. MulticastDelegate 的类，System.MulticastDelegate 又派生自基类 System.Delegate。C#编译器能识别这个类，会使用其委托语法，因此我们不需要了解这个类的具体执行情况。这是 C#与基类共同合作以使编程更易完成的另一个范例。

定义好委托后，就可以创建它的一个实例，从而用该实例存储特定方法的细节。

**注意：**

但是，此处在术语方面有一个问题。类有两个不同的术语："类"表示较广义的定义，"对象"表示类的实例。但委托只有一个术语。在创建委托的实例时，所创建的委托的实例仍称为委托。必须从上下文中确定所使用委托的确切含义。

## 8.2.2　使用委托

下面的代码段说明了如何使用委托。这是在 int 值上调用 ToString()方法的一种相当冗长的方式(代码文件 GetAStringDemo/Program.cs)：

```
private delegate string GetAString();
public static void Main()
{
  int x = 40;
  GetAString firstStringMethod = new GetAString(x.ToString);
  Console.WriteLine($"String is {firstStringMethod()}");
  // With firstStringMethod initialized to x.ToString(),
  // the above statement is equivalent to saying
  // Console.WriteLine($"String is {x.ToString()}");
}
```

在这段代码中，实例化类型为 GetAString 的委托，并对它进行初始化，使其引用整型变量 x 的 ToString()方法。在 C#中，委托在语法上总是接受一个参数的构造函数，这个参数就是委托引用的方法。这个方法必须匹配最初定义委托时的签名。所以在这个示例中，如果用不带参数并返回一个字符串的方法来初始化 firstStringMethod 变量，就会产生一个编译错误。注意，因为 int.ToString()是一个实例方法(不是静态方法)，所以需要指定实例(x)和方法名来正确地初始化委托。

下一行代码使用这个委托来显示字符串。在任何代码中，都应提供委托实例的名称，后面的圆括号中应包含调用该委托中的方法时使用的任何等效参数。所以在上面的代码中，Console.WriteLine()语句完全等价于注释掉的代码行。

实际上，给委托实例提供圆括号与调用委托类的 Invoke()方法完全相同。因为 firstStringMethod 是委托类型的一个变量，所以 C#编译器会用 firstStringMethod.Invoke()代替 firstStringMethod()。

```
firstStringMethod();
firstStringMethod.Invoke();
```

为了减少输入量，在需要委托实例的每个位置可以只传送地址的名称。这称为委托推断。只要编译器可以把委托实例解析为特定的类型，这个 C#特性就是有效的。下面的示例用 GetAString 委托的一个新实例初始化 GetAString 类型的 firstStringMethod 变量：

```
GetAString firstStringMethod = new GetAString(x.ToString);
```

只要用变量 x 将方法名传送给变量 firstStringMethod，就可以编写出作用相同的代码：

```
GetAString firstStringMethod = x.ToString;
```

C#编译器创建的代码是一样的。由于编译器会用 firstStringMethod 检测需要的委托类型，因此它创建 GetAString 委托类型的一个实例，用对象 x 将方法的地址传送给构造函数。

**注意：**

*调用上述方法名时，输入形式不能为 x.ToString()(不要输入圆括号)，也不能把它传送给委托变量。输入圆括号会调用一个方法，而调用 x.ToString()方法会返回一个不能赋予委托变量的字符串对象。只能把方法的地址赋予委托变量。*

委托推断可以在需要委托实例的任何地方使用。委托推断也可以用于事件，因为事件基于委托(参见本章后面的内容)。

委托的一个特征是它们的类型是安全的，可以确保被调用的方法的签名是正确的。但有趣的是，它们不关心在什么类型的对象上调用该方法，甚至不考虑该方法是静态方法还是实例方法。

**注意：**

*给定委托的实例可以引用任何类型的任何对象上的实例方法或静态方法——只要方法的签名匹配委托的签名即可。*

为了说明这一点，扩展上面的代码段，让它使用 firstStringMethod 委托在另一个对象上调用其他两个方法，其中一个是实例方法，另一个是静态方法。为此，使用本章前面定义的 Currency 结构。Currency 结构有自己的 ToString()重载方法和一个与 GetCurrencyUnit()签名相同的静态方法。这样，就可以用同一个委托变量调用这些方法(代码文件 GetAStringDemo/Currency.cs)：

```
struct Currency
{
  public uint Dollars;
  public ushort Cents;
  public Currency(uint dollars, ushort cents)
  {
    Dollars = dollars;
    Cents = cents;
  }

  public override string ToString() => $"${Dollars}.{Cents,2:00}";

  public static string GetCurrencyUnit() => "Dollar";

  public static explicit operator Currency (float value)
  {
    checked
    {
      uint dollars = (uint)value;
      ushort cents = (ushort)((value-dollars) * 100);
      return new Currency(dollars, cents);
    }
  }

  public static implicit operator float (Currency value) =>
    value.Dollars + (value.Cents / 100.0f);

  public static implicit operator Currency (uint value) =>
    new Currency(value, 0);

  public static implicit operator uint (Currency value) =>
    value.Dollars;
}
```

下面可以使用 GetAString 实例，代码如下所示(代码文件 GetAStringDemo/Program.cs)：

```
private delegate string GetAString();
public static void Main()
{
  int x = 40;
  GetAString firstStringMethod = x.ToString;
```

```
  Console.WriteLine($"String is {firstStringMethod()}");
  var balance = new Currency(34, 50);

  // firstStringMethod references an instance method
  firstStringMethod = balance.ToString;
  Console.WriteLine($"String is {firstStringMethod()}");

  // firstStringMethod references a static method
  firstStringMethod = new GetAString(Currency.GetCurrencyUnit);
  Console.WriteLine($"String is {firstStringMethod()}");
}
```

这段代码说明了如何通过委托来调用方法，然后重新给委托指定在类的不同实例上引用的不同方法，甚至可以指定静态方法，或者指定在类的不同类型实例上引用的方法，只要每个方法的签名匹配委托定义即可。

运行此应用程序，会得到委托引用的不同方法的输出结果：

```
String is 40
String is $34.50
String is Dollar
```

但是，我们实际上还没有说明把一个委托传递给另一个方法的具体过程，也没有得到任何特别有用的结果。调用 int 和 Currency 对象的 ToString()方法要比使用委托直观得多！但是，需要用一个相当复杂的示例来说明委托的本质，才能真正领会到委托的用处。下一节会给出两个委托的示例。第一个示例仅使用委托来调用两个不同的操作。它说明了如何把委托传递给方法，如何使用委托数组，但这仍没有很好地说明：没有委托，就不能完成很多工作。第二个示例就复杂得多了，它有一个类 BubbleSorter，该类实现一个方法来按照升序排列一个对象数组。没有委托，就很难编写出这个类。

### 8.2.3 简单的委托示例

在这个示例中，定义一个类 MathOperations，它有两个静态方法，对 double 类型的值执行两种操作。然后使用该委托调用这些方法。MathOperations 类如下所示：

```
class MathOperations
{
  public static double MultiplyByTwo(double value) => value * 2;
  public static double Square(double value) => value * value;
}
```

下面调用这些方法(代码文件 SimpleDelegate/Program.cs)：

```
using System;

namespace Wrox.ProCSharp.Delegates
{
  delegate double DoubleOp(double x);

  class Program
  {
    static void Main()
    {
      DoubleOp[] operations =
      {
        MathOperations.MultiplyByTwo,
        MathOperations.Square
      };

      for (int i=0; i < operations.Length; i++)
      {
        Console.WriteLine($"Using operations[{i}]:);
        ProcessAndDisplayNumber(operations[i], 2.0);
        ProcessAndDisplayNumber(operations[i], 7.94);
        ProcessAndDisplayNumber(operations[i], 1.414);
        Console.WriteLine();
      }
    }

    static void ProcessAndDisplayNumber(DoubleOp action, double value)
    {
      double result = action(value);
      Console.WriteLine($"Value is {value}, result of operation is {result}");
```

```
        }
      }
    }
```

在这段代码中，实例化了一个 DoubleOp 委托的数组(记住，一旦定义了委托类，基本上就可以实例化它的实例，就像处理一般的类那样——所以把一些委托的实例放在数组中是可行的)。该数组的每个元素都初始化为指向由 MathOperations 类实现的不同操作。然后遍历这个数组，把每个操作应用到 3 个不同的值上。这说明了使用委托的一种方式——把方法组合到一个数组中来使用，这样就可以在循环中调用不同的方法了。

这段代码的关键一行是把每个委托实际传递给 ProcessAndDisplayNumber 方法，例如：

```
ProcessAndDisplayNumber(operations[i], 2.0);
```

其中传递了委托名，但不带任何参数。假定 operations[i]是一个委托，在语法上：

- operations[i]表示“这个委托”。换言之，就是委托表示的方法。
- operations[i](2.0)表示“实际上调用这个方法，参数放在圆括号中”。

ProcessAndDisplayNumber 方法定义为把一个委托作为其第一个参数：

```
static void ProcessAndDisplayNumber(DoubleOp action, double value)
```

然后，在这个方法中，调用：

```
double result = action(value);
```

这实际上是调用 action 委托实例封装的方法，其返回结果存储在 result 中。运行这个示例，得到如下所示的结果：

```
SimpleDelegate
Using operations[0]:
Value is 2, result of operation is 4
Value is 7.94, result of operation is 15.88
Value is 1.414, result of operation is 2.828
Using operations[1]:
Value is 2, result of operation is 4
Value is 7.94, result of operation is 63.0436
Value is 1.414, result of operation is 1.999396
```

### 8.2.4　Action<T>和 Func<T>委托

除了为每个参数和返回类型定义一个新委托类型之外，还可以使用 Action<T>和 Func<T>委托。泛型 Action<T>委托表示引用一个 void 返回类型的方法。这个委托类存在不同的变体，可以传递至多 16 种不同的参数类型。没有泛型参数的 Action 类可调用没有参数的方法。Action<in T>调用带一个参数的方法，Action<in T1, in T2>调用带两个参数的方法，Action<in T1, in T2, in T3, in T4, in T5, in T6, in T7, in T8>调用带 8 个参数的方法。

Func<T>委托可以以类似的方式使用。Func<T>允许调用带返回类型的方法。与 Action<T>类似，Func<T>也定义了不同的变体，至多也可以传递 16 个参数类型和一个返回类型。Func<out TResult>委托类型可以调用带返回类型且无参数的方法，Func<in T, out TResult>调用带一个参数的方法，Func<in T1, in T2, in T3, in T4, out TResult>调用带 4 个参数的方法。

8.2.3 节中的示例声明了一个委托，其参数是 double 类型，返回类型是 double：

```
delegate double DoubleOp(double x);
```

除了声明自定义委托 DoubleOp 之外，还可以使用 Func<in T, out TResult>委托。可以声明一个该委托类型的变量，或者声明该委托类型的数组，如下所示：

```
Func<double, double>[] operations =
{
  MathOperations.MultiplyByTwo,
  MathOperations.Square
};
```

使用该委托，并将 ProcessAndDisplayNumber()方法作为参数：

```
static void ProcessAndDisplayNumber(Func<double, double> action,
double value)
{
  double result = action(value);
```

```
  Console.WriteLine($"Value is {value}, result of operation is {result}");
}
```

### 8.2.5 BubbleSorter 示例

下面的示例将说明委托的真正用途。我们要编写一个类 BubbleSorter，它实现一个静态方法 Sort()，这个方法的第一个参数是一个对象数组，把该数组按照升序重新排列。例如，假定传递给该委托的是 int 数组：{0, 5, 6, 2, 1}，则返回的结果应是{0, 1, 2, 5, 6}。

冒泡排序算法非常著名，是一种简单的数字排序方法。它适合于一小组数字，因为对于大量的数字(超过 10 个)，还有更高效的算法。冒泡排序算法重复遍历数组，比较每一对数字，按照需要交换它们的位置，从而把最大的数字逐步移动到数组的末尾。对于给 int 型数字排序，进行冒泡排序的方法如下所示：

```
bool swapped = true;
do
{
  swapped = false;
  for (int i = 0; i < sortArray.Length-1; i++)
  {
    if (sortArray[i] > sortArray[i+1])) // problem with this test
    {
      int temp = sortArray[i];
      sortArray[i] = sortArray[i + 1];
      sortArray[i + 1] = temp;
      swapped = true;
    }
  }
} while (swapped);
```

它非常适合于 int 型，但我们希望 Sort()方法能给任何对象排序。换言之，如果某段客户端代码包含 Currency 结构或自定义的其他类和结构的数组，就需要对该数组排序。这样，上面代码中的 if(sortArray[i] > sortArray[i+1])就有问题了，因为它需要比较数组中的两个对象，看看哪一个更大。可以对 int 型进行这样的比较，但如何对没有实现“>”运算符的新类进行比较？答案是能识别该类的客户端代码必须在委托中传递一个封装的方法，这个方法可以进行比较。另外，不对 temp 变量使用 int 类型，而使用泛型类型就可以实现泛型方法 Sort()。

对于接受类型 T 的泛型方法 Sort<T>()，需要一个比较方法，其两个参数的类型是 T，if 比较的返回类型是布尔类型。这个方法可以从 Func<T1, T2, TResult>委托中引用，其中 T1 和 T2 的类型相同：Func<T, T, bool>。

给 Sort<T>方法指定下述签名：

```
static public void Sort<T>(IList<T> sortArray, Func<T, T, bool> comparison)
```

这个方法的文档声明，comparison 必须引用一个方法，该方法带有两个参数，如果第一个参数的值“小于”第二个参数，就返回 true。

设置完毕后，下面定义 BubbleSorter 类(代码文件 BubbleSorter/BubbleSorter.cs)：

```
class BubbleSorter
{
  static public void Sort<T>(IList<T> sortArray, Func<T, T, bool> comparison)
  {
    bool swapped = true;
    do
    {
      swapped = false;
      for (int i = 0; i < sortArray.Count-1; i++)
      {
        if (comparison(sortArray[i+1], sortArray[i]))
        {
          T temp = sortArray[i];
          sortArray[i] = sortArray[i + 1];
          sortArray[i + 1] = temp;
          swapped = true;
        }
      }
    } while (swapped);
  }
}
```

为了使用这个类，需要定义另一个类，从而建立要排序的数组。在本例中，假定 Mortimer Phones 移动电话公司有一个员工列表，要根据他们的薪水进行排序。每个员工分别由类 Employee 的一个实例表示，如下所示(代码文件 BubbleSorter/Employee.cs)：

```
class Employee
{
  public Employee(string name, decimal salary)
  {
    Name = name;
    Salary = salary;
  }

  public string Name { get; }
  public decimal Salary { get; }

  public override string ToString() => $"{Name}, {Salary:C}";
  public static bool CompareSalary(Employee e1, Employee e2) =>
    e1.Salary < e2.Salary;
}
```

注意，为了匹配 Func<T, T, bool>委托的签名，在这个类中必须定义 CompareSalary，它的参数是两个 Employee 引用，并返回一个布尔值。在实现比较的代码中，根据薪水进行比较。

下面编写一些客户端代码，完成排序(代码文件 BubbleSorter/Program.cs)：

```
using System;

namespace Wrox.ProCSharp.Delegates
{
  class Program
  {
    static void Main()
    {
      Employee[] employees =
      {
        new Employee("Bugs Bunny", 20000),
        new Employee("Elmer Fudd", 10000),
        new Employee("Daffy Duck", 25000),
        new Employee("Wile Coyote", 1000000.38m),
        new Employee("Foghorn Leghorn", 23000),
        new Employee("RoadRunner", 50000)
      };

      BubbleSorter.Sort(employees, Employee.CompareSalary);
      foreach (var employee in employees)
      {
        Console.WriteLine(employee);
      }
    }
  }
}
```

运行这段代码，正确显示按照薪水排列的 Employee，如下所示：

```
Elmer Fudd, $10,000.00
Bugs Bunny, $20,000.00
Foghorn Leghorn, $23,000.00
Daffy Duck, $25,000.00
RoadRunner, $50,000.00
Wile Coyote, $1,000,000.38
```

### 8.2.6　多播委托

前面使用的每个委托都只包含一个方法调用。调用委托的次数与调用方法的次数相同。如果要调用多个方法，就需要多次显式调用这个委托。但是，委托也可以包含多个方法。这种委托称为多播委托。如果调用多播委托，就可以按顺序连续调用多个方法。为此，委托的签名就必须返回 void；否则，就只能得到委托调用的最后一个方法的结果。

可以使用返回类型为 void 的 Action<double>委托(代码文件 MulticastDelegates/Program.cs)：

```
class Program
{
```

```
  static void Main()
  {
    Action<double> operations = MathOperations.MultiplyByTwo;
    operations += MathOperations.Square;
```

在前面的示例中，因为要存储对两个方法的引用，所以实例化了一个委托数组。而这里只是在同一个多播委托中添加两个操作。多播委托可以识别运算符“+”和“+=”。另外，还可以扩展上述代码中的最后两行，如下面的代码段所示：

```
Action<double> operation1 = MathOperations.MultiplyByTwo;
Action<double> operation2 = MathOperations.Square;
Action<double> operations = operation1 + operation2;
```

多播委托还识别运算符“-”和“-=”，以从委托中删除方法调用。

**注意：**

根据后台执行的操作，多播委托实际上是一个派生自 System.MulticastDelegate 的类，System.MulticastDelegate 又派生自基类 System.Delegate。System.MulticastDelegate 的其他成员允许把多个方法调用链接为一个列表。

为了说明多播委托的用法，下面把 SimpleDelegate 示例转换为一个新示例 MulticastDelegate。现在需要委托引用返回 void 的方法，就应重写 MathOperations 类中的方法，让它们显示其结果，而不是返回它们(代码文件 MulticastDelegates/MathOperations.cs)：

```
class MathOperations
{
  public static void MultiplyByTwo(double value)
  {
    double result = value * 2;
    Console.WriteLine($"Multiplying by 2: {value} gives {result}");
  }

  public static void Square(double value)
  {
    double result = value * value;
    Console.WriteLine($"Squaring: {value} gives {result}");
  }
}
```

为了适应这个改变，也必须重写 ProcessAndDisplayNumber()方法(代码文件 MulticastDelegates/Program.cs)：

```
static void ProcessAndDisplayNumber(Action<double> action, double value)
{
  Console.WriteLine();
  Console.WriteLine($"ProcessAndDisplayNumber called with value = {value}");
  action(value);
}
```

下面测试多播委托，其代码如下：

```
static void Main()
{
  Action<double> operations = MathOperations.MultiplyByTwo;
  operations += MathOperations.Square;
  ProcessAndDisplayNumber(operations, 2.0);
  ProcessAndDisplayNumber(operations, 7.94);
  ProcessAndDisplayNumber(operations, 1.414);
  Console.WriteLine();
}
```

现在，每次调用 ProcessAndDisplayNumber()方法时，都会显示一条消息，说明它已经被调用。然后，下面的语句会按顺序调用 action 委托实例中的每个方法：

```
action(value);
```

运行这段代码，得到如下所示的结果：

```
ProcessAndDisplayNumber called with value = 2
Multiplying by 2: 2 gives 4
Squaring: 2 gives 4

ProcessAndDisplayNumber called with value = 7.94
Multiplying by 2: 7.94 gives 15.88
```

```
Squaring: 7.94 gives 63.0436

ProcessAndDisplayNumber called with value = 1.414
Multiplying by 2: 1.414 gives 2.828
Squaring: 1.414 gives 1.999396
```

如果正在使用多播委托，就应知道对同一个委托，调用其方法链的顺序并未正式定义。因此应避免编写依赖于以特定顺序调用方法的代码。

通过一个委托调用多个方法还可能导致一个更严重的问题。多播委托包含一个逐个调用的委托集合。如果通过委托调用的其中一个方法抛出一个异常，整个迭代就会停止。下面是 MulticastIteration 示例，其中定义了一个简单的委托 Action，它没有参数并返回 void。这个委托打算调用 One()和 Two()方法，这两个方法满足委托的参数和返回类型要求。注意 One()方法抛出了一个异常(代码文件 MulticastDelegateWithIteration/Program.cs)：

```
using System;

namespace Wrox.ProCSharp.Delegates
{
  class Program
  {
    static void One()
    {
      Console.WriteLine("One");
      throw new Exception("Error in one");
    }

    static void Two()
    {
      Console.WriteLine("Two");
    }
```

在 Main()方法中，创建了委托 d1，它引用方法 One()；接着把 Two()方法的地址添加到同一个委托中。调用 d1 委托，就可以调用这两个方法。在 try/catch 块中捕获异常：

```
static void Main()
{
  Action d1 = One;
  d1 += Two;
  try
  {
    d1();
  }
  catch (Exception)
  {
    Console.WriteLine("Exception caught");
  }
}
```

委托只调用了第一个方法。因为第一个方法抛出了一个异常，所以委托的迭代会停止，不再调用 Two()方法。没有指定调用方法的顺序时，结果会有所不同。

```
One
Exception Caught
```

**注意：**
错误和异常的介绍详见第 14 章。

在这种情况下，为了避免这个问题，应自己迭代方法列表。Delegate 类定义 GetInvocationList()方法，它返回一个 Delegate 对象数组。现在可以使用这个委托调用与委托直接相关的方法，捕获异常，并继续下一次迭代(代码文件 MulticastDelegatesUsingInvocationList/Program.cs)：

```
static void Main()
{
  Action d1 = One;
  d1 += Two;
  Delegate[] delegates = d1.GetInvocationList();
  foreach (Action d in delegates)
  {
    try
    {
```

```
      d();
    }
    catch (Exception)
    {
      Console.WriteLine("Exception caught");
    }
  }
}
```

修改了代码后，运行应用程序，会看到在捕获了异常后将继续迭代下一个方法。

```
One
Exception caught
Two
```

### 8.2.7 匿名方法

到目前为止，要想使委托工作，方法必须已经存在(即委托通过其将调用方法的相同签名定义)。但还有另外一种使用委托的方式：通过匿名方法。匿名方法是用作委托的参数的一段代码。

用匿名方法定义委托的语法与前面的定义并没有区别。但在实例化委托时，就会出现区别。下面是一个非常简单的控制台应用程序，它说明了如何使用匿名方法(代码文件 AnonymousMethods/Program.cs)：

```
class Program
{
  static void Main()
  {
    string mid = ", middle part,";
    Func<string, string> anonDel = delegate(string param)
    {
      param += mid;
      param += " and this was added to the string.";
      return param;
    };
    Console.WriteLine(anonDel("Start of string"));
  }
}
```

Func<string, string>委托接受一个字符串参数，返回一个字符串。anonDel 是这种委托类型的变量。不是把方法名赋予这个变量，而是使用一段简单的代码：前面是关键字 delegate，后面是一个字符串参数。

可以看出，该代码块使用方法级的字符串变量 mid，该变量是在匿名方法的外部定义的，并将其添加到要传递的参数中。接着代码返回该字符串值。在调用委托时，把一个字符串作为参数传递，将返回的字符串输出到控制台上。

使用匿名方法的优点是减少了要编写的代码。不必定义仅由委托使用的方法。在为事件定义委托时，这一点非常明显(本章后面探讨事件)。这有助于降低代码的复杂性，尤其是在定义了好几个事件时，代码会显得比较简单。使用匿名方法时，代码执行速度并没有加快。编译器仍定义了一个方法，该方法只有一个自动指定的名称，我们不需要知道这个名称。

在使用匿名方法时，必须遵循两条规则。在匿名方法中不能使用跳转语句(break、goto 或 continue)跳到该匿名方法的外部，反之亦然：匿名方法外部的跳转语句不能跳到该匿名方法的内部。

在匿名方法内部不能访问不安全的代码。另外，也不能访问在匿名方法外部使用的 ref 和 out 参数。但可以使用在匿名方法外部定义的其他变量。

如果需要用匿名方法多次编写同一个功能，就不要使用匿名方法。此时与复制代码相比，编写一个命名方法比较好，因为该方法只需要编写一次，以后可通过名称引用它。

**注意：**

匿名方法的语法在 C# 2 中引入。在新的程序中，并不需要这个语法，因为 lambda 表达式(参见下一节)提供了相同的功能，还提供了其他功能。但是，在已有的源代码中，许多地方都使用了匿名方法，所以最好了解它。

从 C# 3.0 开始，可以使用 lambda 表达式。

# 8.3 lambda 表达式

使用 lambda 表达式的一个场合是把 lambda 表达式赋予委托类型：在线实现代码。只要有委托参数类型的地方，就可以使用 lambda 表达式。前面使用匿名方法的例子可以改为使用 lambda 表达式。

```
class Program
{
  static void Main()
  {
    string mid = ", middle part,";
    Func<string, string> lambda = param =>
    {
      param += mid;
      param += " and this was added to the string.";
      return param;
    };
    Console.WriteLine(lambda("Start of string"));
  }
}
```

lambda 运算符“=>”的左边列出了需要的参数，而其右边定义了赋予 lambda 变量的方法的实现代码。

## 8.3.1 参数

lambda 表达式有几种定义参数的方式。如果只有一个参数，只写出参数名就足够了。下面的 lambda 表达式使用了参数 s。因为委托类型定义了一个 string 参数，所以 s 的类型就是 string。实现代码调用 String.Format()方法来返回一个字符串，在调用该委托时，就把该字符串最终写入控制台(代码文件 LambdaExpressions/Program.cs)：

```
Func<string, string> oneParam = s => $"change uppercase {s.ToUpper()}";
Console.WriteLine(oneParam("test"));
```

如果委托使用多个参数，就把这些参数名放在圆括号中。这里参数 x 和 y 的类型是 double，由 Func<double, double, double>委托定义：

```
Func<double, double, double> twoParams = (x, y) => x * y;
Console.WriteLine(twoParams(3, 2));
```

为了方便起见，可以在圆括号中给变量名添加参数类型。如果编译器不能匹配重载后的版本，那么使用参数类型可以帮助找到匹配的委托：

```
Func<double, double, double> twoParamsWithTypes =
  (double x, double y) => x * y;
Console.WriteLine(twoParamsWithTypes(4, 2));
```

## 8.3.2 多行代码

如果 lambda 表达式只有一条语句，在方法块内就不需要花括号和 return 语句，因为编译器会添加一条隐式的 return 语句：

```
Func<double, double> square = x => x * x;
```

添加花括号、return 语句和分号是完全合法的，通常这比不添加这些符号更容易阅读：

```
Func<double, double> square = x =>
{
  return x * x;
}
```

但是，如果在 lambda 表达式的实现代码中需要多条语句，就必须添加花括号和 return 语句：

```
Func<string, string> lambda = param =>
{
  param += mid;
  param += " and this was added to the string.";
  return param;
};
```

### 8.3.3 闭包

通过 lambda 表达式可以访问 lambda 表达式块外部的变量，这称为闭包。闭包是非常好用的功能，但如果使用不当，也会非常危险。

在下面的示例中，Func<int, int>类型的 lambda 表达式需要一个 int 参数，返回一个 int 值。该 lambda 表达式的参数用变量 x 定义。实现代码还访问了 lambda 表达式外部的变量 someVal。只要不假设在调用 f 时，lambda 表达式创建了一个以后使用的新方法，这似乎没有什么问题。看看下面这个代码块，调用 f 的返回值应是 x 加 5 的结果，但实情似乎不是这样(代码文件 LambdaExpressions/Program.cs)：

```
int someVal = 5;
Func<int, int> f = x => x + someVal;
```

假定以后要修改变量 someVal，于是调用 lambda 表达式时，会使用 someVal 的新值。调用 f(3)的结果是 10：

```
someVal = 7;
WriteLine(f(3));
```

同样，在 lambda 表达式中修改闭包的值时，可以在 lambda 表达式外部访问已改动的值。

现在我们也许会奇怪，如何在 lambda 表达式的内部访问 lambda 表达式外部的变量。为了理解这一点，看看编译器在定义 lambda 表达式时做了什么。对于 lambda 表达式 x => x + someVal，编译器会创建一个匿名类，它有一个构造函数来传递外部变量。该构造函数取决于从外部访问的变量数。对于这个简单的例子，构造函数接受一个 int 值。匿名类包含一个匿名方法，其实现代码、参数和返回类型由 lambda 表达式定义：

```
public class AnonymousClass
{
  private int someVal;
  public AnonymousClass(int someVal)
  {
    this.someVal = someVal;
  }
  public int AnonymousMethod(int x) => x + someVal;
}
```

使用 lambda 表达式并调用该方法，会创建匿名类的一个实例，并传递调用该方法时变量的值。

**注意：**
如果给多个线程使用闭包，就可能遇到并发冲突。最好仅给闭包使用不变的类型。这样可以确保不改变值，也不需要同步。

**注意：**
lambda 表达式可以用于类型为委托的任意地方。类型是 Expression 或 Expression<T>时，也可以使用 lambda 表达式，此时编译器会创建一个表达式树。该功能的介绍详见第 12 章。

## 8.4 事件

事件基于委托，为委托提供了一种发布/订阅机制。在.NET 架构内到处都能看到事件。在 Windows 应用程序中，Button 类提供了 Click 事件。这类事件就是委托。触发 Click 事件时调用的处理程序方法需要得到定义，而其参数由委托类型定义。

在本节的示例代码中，事件用于连接 CarDealer 类和 Consumer 类。CarDealer 类提供了一个新车到达时触发的事件。Consumer 类订阅该事件，以获得新车到达的通知。

### 8.4.1 事件发布程序

我们从 CarDealer 类开始介绍，它基于事件提供一个订阅。CarDealer 类用 event 关键字定义了类型为 EventHandler<CarInfoEventArgs>的 NewCarInfo 事件。在 NewCar()方法中，通过调用 RaiseNewCarInfo 方法触发

NewCarInfo 事件。这个方法的实现确认委托是否为空，如果不为空，就引发事件(代码文件 EventSample/CarDealer.cs)：

```
using System;
namespace Wrox.ProCSharp.Delegates
{
  public class CarInfoEventArgs: EventArgs
  {
    public CarInfoEventArgs(string car) => Car = car;

    public string Car { get; }
  }

  public class CarDealer
  {
    public event EventHandler<CarInfoEventArgs> NewCarInfo;
    public void NewCar(string car)
    {
      Console.WriteLine($"CarDealer, new car {car}");
      NewCarInfo?.Invoke(this, new CarInfoEventArgs(car));
    }
  }
}
```

**注意：**

*前面例子中使用的空传播运算符.?是 C# 6 新增的运算符。这个运算符的讨论参见第 6 章。*

CarDealer 类提供了 EventHandler<CarInfoEventArgs>类型的 NewCarInfo 事件。作为一个约定，事件一般使用带两个参数的方法；其中第一个参数是一个对象，包含事件的发送者，第二个参数提供了事件的相关信息。第二个参数随不同的事件类型而改变。.NET 1.0 为所有不同数据类型的事件定义了几百个委托。有了泛型委托 EventHandler<T>后，就不再需要委托了。EventHandler<TEventArgs>定义了一个处理程序，它返回 void，接受两个参数。对于 EventHandler<TEventArgs>，第一个参数必须是 object 类型，第二个参数是 T 类型。EventHandler<TEventArgs>还定义了一个关于 T 的约束；它必须派生自基类 EventArgs，CarInfoEventArgs 就派生自基类 EventArgs：

```
public event EventHandler<CarInfoEventArgs> NewCarInfo;
```

委托 EventHandler<TEventArgs>的定义如下：

```
public delegate void EventHandler<TEventArgs>(object sender, TEventArgs e)
where TEventArgs: EventArgs
```

在一行上定义事件是 C#的简化记法。编译器会创建一个 EventHandler<CarInfoEventArgs>委托类型的变量，并添加方法，以便从委托中订阅和取消订阅。该简化记法的较长形式如下所示。这非常类似于自动属性和完整属性之间的关系。对于事件，使用 add 和 remove 关键字添加和删除委托的处理程序：

```
private EventHandler<CarInfoEventArgs> _newCarInfo;
public event EventHandler<CarInfoEventArgs> NewCarInfo
{
  add => _newCarInfo += value;
  remove => _newCarInfo -= value;
}
```

**注意：**

*如果不仅需要添加和删除事件处理程序，定义事件的长记法就很有用，例如，需要为多个线程访问添加同步操作。WPF 控件使用长记法给事件添加冒泡和隧道功能。*

CarDealer 类通过调用委托的 Invoke 方法触发事件。可以调用给事件订阅的所有处理程序。注意，与之前的多播委托一样，方法的调用顺序无法保证。为了更多地控制处理程序的调用，可以使用 Delegate 类的 GetInvocationList()方法，访问委托列表中的每一项，并独立地调用每个方法，如上所示。

```
NewCarInfo?.Invoke(this, new CarInfoEventArgs(car));
```

触发事件是只包含一行代码的程序。然而，这只是C# 6的功能。在C# 6版本之前，触发事件会更复杂。这是C# 6之前实现的相同功能。在触发事件之前，需要检查事件是否为空。因为在进行null检查和触发事件之间，可以使用另一个线程把事件设置为null，所以使用一个局部变量，如下所示：

```
EventHandler<CarInfoEventArgs> newCarInfo = NewCarInfo;
if (newCarInfo != null)
{
  newCarInfo(this, new CarInfoEventArgs(car));
}
```

在C# 6中，所有这一切都可以使用null传播运算符和一个代码行取代，如前所示。

在触发事件之前，需要检查委托NewCarInfo是否不为空。如果没有订阅处理程序，委托就为空：

```
protected virtual void RaiseNewCarInfo(string car)
{
  NewCarInfo?.Invoke(this, new CarInfoEventArgs(car));
}
```

### 8.4.2 事件侦听器

Consumer类用作事件侦听器。这个类订阅了CarDealer类的事件，并定义了NewCarIsHere方法，该方法满足EventHandler<CarInfoEventArgs>委托的要求，该委托的参数类型是object和CarInfoEventArgs(代码文件EventsSample/Consumer.cs)：

```
public class Consumer
{
  private string _name;
  public Consumer(string name) => _name = name;

  public void NewCarIsHere(object sender, CarInfoEventArgs e)
  {
    Console.WriteLine($"{_name}: car {e.Car} is new");
  }
}
```

现在需要连接事件发布程序和订阅器。为此使用CarDealer类的NewCarInfo事件，通过“+=”创建一个订阅。消费者Valtteri订阅了事件，接着消费者Max也订阅了事件，然后Valtteri通过“-=”取消了订阅(代码文件EventsSample/Program.cs)。

```
class Program
{
  static void Main()
  {
    var dealer = new CarDealer();
    var valtteri = new Consumer("Valtteri");
    dealer.NewCarInfo += valtteri.NewCarIsHere;
    dealer.NewCar("Williams");

    var max = new Consumer("Max");
    dealer.NewCarInfo += max.NewCarIsHere;
    dealer.NewCar("Mercedes");
    dealer.NewCarInfo -= valtteri.NewCarIsHere;
    dealer.NewCar("Ferrari");
  }
}
```

运行应用程序，一辆Williams汽车到达，Valtteri得到了通知。因为之后Max也注册了该订阅，所以Valtteri和Max都获得了新款Mercedes汽车的通知。接着Valtteri取消了订阅，所以只有Max获得了Ferrari汽车的通知：

```
CarDealer, new car Williams
Valtteri: car Williams is new
CarDealer, new car Mercedes
Valtteri: car Mercedes is new
Max: car Mercedes is new
CarDealer, new car Ferrari
Max: car Ferrari is new
```

## 8.5　小结

本章介绍了委托、lambda 表达式和事件的基础知识，解释了如何声明委托，如何给委托列表添加方法，如何实现通过委托和 lambda 表达式调用的方法，并讨论了声明事件处理程序来响应事件的过程，以及如何创建自定义事件，使用引发事件的模式。

在设计大型应用程序时，使用委托和事件可以减少依赖性和各层的耦合，并能开发出具有更高重用性的组件。

lambda 表达式是基于委托的 C#语言特性，通过它们可以减少需要编写的代码量。lambda 表达式不仅仅用于委托，也用于 LINQ(详见第 12 章)。

第 9 章介绍字符串和正则表达式的使用。

# 第 9 章

# 字符串和正则表达式

**本章要点**

- 创建字符串
- 格式化表达式
- 使用正则表达式
- 使用 Span<T>和字符串

**本章源代码下载地址(wrox.com)：**

打开 www.wrox.com 的 Download Code 选项卡可下载本章源代码。源代码也可以在 StringsAndRegularExpressions 目录的 https://github.com/ProfessionalCSharp/ProfessionalCSharp7 中找到。本章代码分为以下几个主要的示例文件：

- StringSample
- StringFormats
- RegularExpressionPlayground
- SpanWithStrings

从本书一开始，我们就在使用字符串，因为每个程序都需要字符串。但读者可能没有意识到，在 C#中 string 关键字的映射实际上指向.NET 基类 System.String。System.String 是一个功能非常强大且用途广泛的基类，但它不是.NET 库中唯一与字符串相关的类。本章首先复习一下 System.String 的特性，再介绍如何使用其他的.NET 库类来处理字符串，特别是 System.Text 和 System.Text.RegularExpressions 名称空间中的类。本章主要介绍下述内容：

- 构建字符串——如果多次修改一个字符串，例如，创建一个长字符串，然后显示该字符串或将其传递给其他方法或应用程序，String 类就会变得效率低下。对于这种情况，应使用另一个类 System.Text.StringBuilder，因为它是专门为这种情况设计的。
- 格式化表达式——这些格式化表达式将用于后面几章中的 Console.WriteLine()方法。格式化表达式使用两个有用的接口 IFormatProvider 和 IFormattable 来处理。在自己的类上实现这两个接口，实际上就可以定义自己的格式化序列，这样，Console.WriteLine()和类似的类就可以按指定的方式显示类的值。
- 正则表达式——.NET 还提供了一些非常复杂的类来识别字符串，或从长字符串中提取满足某些复杂条件的子字符串。例如，找出字符串中所有重复出现的某个字符或一组字符，或者找出以 s 开头且至少包含一个 n 的所有单词，又或者找出遵循雇员 ID 或社会安全号码结构的字符串。虽然可以使用 String 类，

编写方法来完成这类处理，但这类方法编写起来比较烦琐。而使用 System.Text.RegularExpressions 名称空间中的类就比较简单，System.Text. RegularExpressions 专门用于完成这类处理。

- Span——.NET Core 提供了泛型 Span 结构，它允许快速访问内存。Span<T>允许访问字符串的切片，而不需要复制字符串。

## 9.1　System.String 类

在介绍其他字符串类之前，先快速复习一下 String 类中一些可用的方法。

System.String 类专门用于存储字符串，允许对字符串进行许多操作。此外，由于这种数据类型非常重要，C#提供了它自己的关键字和相关的语法，以便使用这个类来轻松地处理字符串。

使用运算符重载可以连接字符串：

```
string message1 = "Hello"; // returns "Hello"
message1 += ", There"; // returns "Hello, There"
string message2 = message1 + "!"; // returns "Hello, There!"
```

C#还允许使用类似于索引器的语法来提取指定的字符：

```
string message = "Hello";
char char4 = message[4]; // returns 'o'. Note the string is zero-indexed
```

这个类可以完成许多常见的任务，如替换字符、删除空白和把字母变成大写形式等。可用的方法如表 9-1 所示。

表 9-1

| 方　法 | 作　用 |
|---|---|
| Compare | 比较字符串的内容，考虑区域值背景(区域设置)，判断某些字符是否相等 |
| CompareOrdinal | 与 Compare 一样，但不考虑区域值背景 |
| Concat | 把多个字符串实例合并为一个实例 |
| CopyTo | 把从选定下标开始的特定数量字符复制到数组的一个全新实例中 |
| Format | 格式化包含各种值的字符串和如何格式化每个值的说明符 |
| IndexOf | 定位字符串中第一次出现某个给定子字符串或字符的位置 |
| IndexOfAny | 定位字符串中第一次出现某个字符或一组字符的位置 |
| Insert | 把一个字符串实例插入到另一个字符串实例的指定索引处 |
| Join | 合并字符串数组，创建一个新字符串 |
| LastIndexOf | 与 IndexOf 一样，但定位最后一次出现的位置 |
| LastIndexOfAny | 与 IndexOfAny 一样，但定位最后一次出现的位置 |
| PadLeft | 在字符串的左侧，通过添加指定的重复字符填充字符串 |
| PadRight | 在字符串的右侧，通过添加指定的重复字符填充字符串 |
| Replace | 用另一个字符或子字符串替换字符串中给定的字符或子字符串 |
| Split | 在出现给定字符的地方，把字符串拆分为一个子字符串数组 |
| Substring | 在字符串中检索给定位置的子字符串 |
| ToLower | 把字符串转换为小写形式 |
| ToUpper | 把字符串转换为大写形式 |
| Trim | 删除首尾的空白 |

**注意：**
表 9-1 并不完整，但可以让你明白字符串所提供的功能。

### 9.1.1　构建字符串

如上所述，String 类是一个功能非常强大的类，它实现许多很有用的方法。但是，String 类存在一个问题：重复修改给定的字符串，效率会很低，它实际上是一个不可变的数据类型，这意味着一旦对字符串对象进行了初始化，该字符串对象就不能改变了。表面上修改字符串内容的方法和运算符实际上是创建一个新字符串，根据需要，可以把旧字符串的内容复制到新字符串中。例如，考虑下面的代码(代码文件 StringSample/Program.cs)：

```
string greetingText = "Hello from all the people at Wrox Press. ";
greetingText += "We do hope you enjoy this book as much as we enjoyed writing it.";
```

在执行这段代码时，首先创建一个 System.String 类型的对象，并把它初始化为文本“Hello from all the people at Wrox Press. ”，注意句号后面有一个空格。此时.NET 运行库会为该字符串分配足够的内存来保存这个文本(41 个字符)，再设置变量 greetingText 来表示这个字符串实例。

从语法上看，下一行代码是把更多的文本添加到字符串中。实际上并非如此，在此是创建一个新字符串实例，给它分配足够的内存，以存储合并的文本(共 104 个字符)。把最初的文本“Hello from all the people at Wrox Press. ”复制到这个新字符串中，再加上额外的文本“We do hope you enjoy this book as much as we enjoyed writing it.”。然后更新存储在变量 greetingText 中的地址，使变量正确地指向新的字符串对象。现在没有引用旧的字符串对象——不再有变量引用它，下一次垃圾收集器清理应用程序中所有未使用的对象时，就会删除它。

这段代码本身还不错，但假定要对这个字符串编码，将其中的每个字符的 ASCII 值加 1，形成非常简单的加密模式。这就会把该字符串变成“Ifmmp gspn bmm uif hvst bu Xspy Qsftt. Xf ep ipqf zpv fokpz uijt cppl bt nvdi bt xf fokpzfe xsjujoh ju”。完成这个任务有好几种方式，但最简单、最高效的一种(假定只使用 String 类)是使用 String.Replace()方法，该方法把字符串中指定的子字符串用另一个子字符串代替。使用 Replace()，对文本进行编码的代码如下所示：

```
string greetingText = "Hello from all the people at Wrox Press. ";
greetingText += "We do hope you enjoy this book as much as we " +
"enjoyed writing it.";
Console.WriteLine($"Not encoded:\n {greetingText}");
for(int i = 'z'; i>= 'a'; i--)
{
  char old1 = (char)i;
  char new1 = (char)(i+1);
  greetingText = greetingText.Replace(old1, new1);
}

for(int i = 'Z'; i>='A'; i--)
{
  char old1 = (char)i;
  char new1 = (char)(i+1);
  greetingText = greetingText.Replace(old1, new1);
}
Console.WriteLine($"Encoded:\n {greetingText}");
```

**注意：**
为了简单起见，这段代码没有把 Z 换成 A，也没有把 z 换成 a。这些字符分别编码为[和{。

在本示例中，Replace()方法以一种智能的方式工作，在某种程度上，它并没有创建一个新字符串，除非其实际上要对旧字符串进行某些改变。原来的字符串包含 23 个不同的小写字母和 3 个不同的大写字母。所以 Replace()分配一个新字符串，共计分配 26 次，每个新字符串都包含 103 个字符。因此加密过程需要在堆上有一个总共能存储 2678 个字符的字符串对象，该对象最终将等待被垃圾收集！显然，如果使用字符串频繁进行文字处理，应用程序就会遇到严重的性能问题。

为了解决这类问题，Microsoft 提供了 System.Text.StringBuilder 类，StringBuilder 类不像 String 类那样能够

支持非常多的方法。在 StringBuilder 类上可以进行的处理仅限于替换和追加或删除字符串中的文本。但是，它的工作方式非常高效。

在使用 String 类构造一个字符串时，要给它分配足够的内存来保存字符串。然而，StringBuilder 类通常分配的内存会比它需要的更多。开发人员可以选择指定 StringBuilder 要分配多少内存，但如果没有指定，在默认情况下就根据初始化 StringBuilder 实例时的字符串长度来确定所用内存的大小。StringBuilder 类有两个主要的属性：

- Length——指定包含字符串的实际长度。
- Capacity——指定字符串在分配的内存中的最大长度。

对字符串的修改就在赋予 StringBuilder 实例的内存块中进行，这就大大提高了追加子字符串和替换单个字符的效率。删除或插入子字符串仍然效率低下，因为这需要移动随后的字符串部分。只有执行扩展字符串容量的操作时，才需要给字符串分配新内存，这样才能移动包含的整个字符串。在添加额外的容量时，从经验来看，如果 StringBuilder 类检测到容量超出，且没有设置新值，就会使自己的容量翻倍。

例如，如果使用 StringBuilder 对象构造最初的欢迎字符串，就可以编写下面的代码：

```
var greetingBuilder =
  new StringBuilder("Hello from all the people at Wrox Press. ", 150);
greetingBuilder.Append("We do hope you enjoy this book as much " +
  "as we enjoyed writing it");
```

**注意：**
*为了使用 StringBuilder 类，需要在代码中引用 System.Text 类。*

在这段代码中，为 StringBuilder 类设置的初始容量是 150。最好把容量设置为字符串可能的最大长度，确保 StringBuilder 类不需要重新分配内存，因为其容量足够用了。该容量默认设置为 16。理论上，可以设置尽可能大的数字，足够给该容量传送一个 int 值，但如果实际上给字符串分配 20 亿个字符的空间(这是 StringBuilder 实例理论上允许拥有的最大空间)，系统就可能会没有足够的内存。

然后，在调用 AppendFormat()方法时，其他文本就放在空的空间中，不需要分配更多的内存。但是，多次替换文本才能获得使用 StringBuilder 类所带来的高效性能。例如，如果要以前面的方式加密文本，就可以执行整个加密过程，不需要分配更多的内存：

```
var greetingBuilder =
  new StringBuilder("Hello from all the people at Wrox Press. ", 150);
greetingBuilder.AppendFormat("We do hope you enjoy this book as much " +
  "as we enjoyed writing it");
Console.WriteLine("Not Encoded:\n" + greetingBuilder);
for(int i = 'z'; i>='a'; i--)
{
  char old1 = (char)i;
  char new1 = (char)(i+1);
  greetingBuilder = greetingBuilder.Replace(old1, new1);
}

for(int i = 'Z'; i>='A'; i--)
{
  char old1 = (char)i;
  char new1 = (char)(i+1);
  greetingBuilder = greetingBuilder.Replace(old1, new1);
}
Console.WriteLine($"Encoded:\n {greetingBuilder}");
```

这段代码使用了 StringBuilder.Replace()方法，它的功能与 String.Replace()一样，但不需要在过程中复制字符串。在上述代码中，为存储字符串而分配的总存储单元是用于 StringBuilder 实例的 150 个字符，以及在最后一条 Console.WriteLine()语句中执行字符串操作期间分配的内存。

一般而言，使用 StringBuilder 类执行字符串的任何操作，而使用 String 类存储字符串或显示最终结果。

### 9.1.2　StringBuilder 成员

前面介绍了 StringBuilder 类的一个构造函数，它的参数是一个初始字符串及该字符串的容量。StringBuilder 类还有几个其他的构造函数。例如，可以只提供一个字符串：

```
var sb = new StringBuilder("Hello");
```

或者用给定的容量创建一个空的 StringBuilder 类：

```
var sb = new StringBuilder(20);
```

除了前面介绍的 Length 和 Capacity 属性外，还有一个只读属性 MaxCapacity，它表示对给定的 StringBuilder 实例的容量限制。在默认情况下，这由 int.MaxValue 给定(大约 20 亿，如前所述)。但在构造 StringBuilder 对象时，也可以把这个值设置为较低的值：

```
// This will set the initial capacity to 100, but the max will be 500.
// Hence, this StringBuilder can never grow to more than 500 characters,
// otherwise it will raise an exception if you try to do that.
var sb = new StringBuilder(100, 500);
```

还可以随时显式地设置容量，但如果把这个值设置为小于字符串的当前长度，或者是超出了最大容量的某个值，就会抛出一个异常：

```
var sb = new StringBuilder("Hello");
sb.Capacity = 100;
```

StringBuilder 类主要的方法如表 9-2 所示。

表 9-2

| 方　法 | 说　明 |
|---|---|
| Append() | 给当前字符串追加一个字符串 |
| AppendFormat() | 追加特定格式的字符串 |
| Insert() | 在当前字符串中插入一个子字符串 |
| Remove() | 从当前字符串中删除字符 |
| Replace() | 在当前字符串中，用某个字符全部替换另一个字符，或者用当前字符串中的一个子字符串全部替换另一个字符串 |
| ToString() | 返回当前强制转换为 System.String 对象的字符串(在 System.Object 中重写) |

其中一些方法还有几种重载版本。

**注意：**
AppendFormat()方法实际上会在最终调用 Console.WriteLine()方法时被调用，它负责确定所有像{0:D}的格式化表达式应使用什么表达式替代。下一节讨论这个问题。

不能把 StringBuilder 强制转换为 String(隐式转换和显式转换都不行)。如果要把 StringBuilder 的内容输出为 String，唯一的方式就是使用 ToString()方法。

前面介绍了 StringBuilder 类，说明了使用它提高性能的一些方式。但要注意，这个类并不总能提高性能。StringBuilder 类基本上应在处理多个字符串时使用。但如果只是连接两个字符串，使用 System.String 类会较好。

## 9.2　字符串格式

之前的章节介绍了用$前缀给字符串传递变量。本章讨论这个 C#功能背后的理论，并囊括格式化字符串提供的所有其他功能。

## 9.2.1　字符串插值

C# 6 引入了给字符串使用$前缀的字符串插值。下面的示例使用$前缀创建了字符串 s2，这个前缀允许在花括号中包含占位符来引用代码的结果。{ s1 }是字符串中的一个占位符，编译器将变量 s1 的值放在字符串 s2 中(代码文件 StringFormats/Program.cs)：

```
string s1 = "World";
string s2 = $"Hello, {s1}";
```

在现实中，这只是语法糖。对于带$前缀的字符串，编译器创建 String.Format 方法的调用。所以前面的代码段解读为：

```
string s1 = "World";
string s2 = String.Format("Hello, {0}", s1);
```

String.Format 方法的第一个参数接受一个格式字符串，其中的占位符从 0 开始编号，其后是放入字符串空白处的参数。

新的字符串格式要方便得多，不需要编写那么多代码。

不仅可以使用变量来填写字符串的空白处，还可以使用返回一个值的任何方法：

```
string s2 = $"Hello, {s1.ToUpper()}";
```

这段代码可解读为如下类似的语句：

```
string s2 = String.Format("Hello, {0}", s1.ToUpper());
```

字符串还可以有多个空白处，如下所示的代码：

```
int x = 3, y = 4;
string s3 = $"The result of {x} + {y} is {x + y}";
```

解读为：

```
string s3 = String.Format("The result of {0} and {1} is {2}", x, y, x + y);
```

### 1. FormattableString

把字符串赋予 FormattableString，就很容易得到翻译过来的插值字符串。插值字符串可以直接分配，因为 FormattableString 比正常的字符串更适合匹配。这个类型定义了 Format 属性(返回得到的格式字符串)、ArgumentCount 属性和方法 GetArgument(返回值)：

```
int x = 3, y = 4;
FormattableString s = $"The result of {x} + {y} is {x + y}";
Console.WriteLine($"format: {s.Format}");
for (int i = 0; i < s.ArgumentCount; i++)
{
  Console.WriteLine($"argument {i}: {s.GetArgument(i)}");
}
```

运行此代码段，输出结果如下：

```
format: The result of {0} + {1} is {2}
argument 0: 3
argument 1: 4
argument 2: 7
```

**注意：**

类 FormattableString 在 System 名称空间中定义，但是需要.NET 4.6。如果想在.NET 旧版本中使用 FormattableString，可以自己创建这种类型，或使用 NuGet 包 StringInterpolationBridge。

### 2. 给字符串插值使用其他区域值

插值字符串默认使用当前的区域值，这很容易改变。辅助方法 Invariant 把插值字符串改为使用不变的区域值，而不是当前的区域值。因为插值字符串可以分配给 FormattableString 类型，所以它们可以传递给这个方法。FormattableString 定义了允许传递 IFormatProvider 的 ToString 方法。接口 IFormatProvider 由 CultureInfo 类实现。

把 CultureInfo.InvariantCulture 传递给 IFormatProvider 参数，就可把字符串改为使用不变的区域值：

```
private string Invariant(FormattableString s) =>
  s.ToString(CultureInfo.InvariantCulture);
```

**注意：**
第 27 章讨论了格式字符串的语言专有问题，以及区域值和不变的区域值。

在下面的代码段中，Invariant 方法用来把一个字符串传递给第二个 WriteLine 方法。WriteLine 的第一个调用使用当前的区域值，而第二个调用使用不变的区域值：

```
var day = new DateTime(2025, 2, 14);
Console.WriteLine($"{day:d}");
Console.WriteLine(Invariant($"{day:d}"));
```

如果有英语区域值设置，结果就如下所示。如果系统配置了另一个区域值，第一个结果就是不同的。在任何情况下，都会看到不变区域值的差异：

```
2/14/2025
02/14/2015
```

使用不变的区域值，不需要自己实现方法，而可以直接使用 FormattableString 类的静态方法 Invariant：

```
Console.WriteLine(FormattableString.Invariant($"{day:d}"));
```

### 3. 转义花括号

如果希望在插值字符串中包括花括号，可以使用两个花括号转义它们：

```
string s = "Hello";
Console.WriteLine($"{{s}} displays the value of s: {s}");
```

WriteLine 方法被解读为如下实现代码：

```
Console.WriteLine(String.Format("{s} displays the value of s: {0}", s));
```

输出如下：

```
{s} displays the value of s : Hello
```

还可以转义花括号，从格式字符串中建立一个新的格式字符串。下面看看这个代码段：

```
string formatString = $"{s}, {{0}}";
string s2 = "World";
WriteLine(formatString, s2);
```

有了字符串变量 formatString，编译器会把占位符 0 插入变量 s，调用 String.Format：

```
string formatString = String.Format("{0}, {{0}}", s);
```

这会生成格式字符串，其中变量 s 替换为值 Hello，删除第二个格式最外层的花括号：

```
string formatString = "Hello, {0}";
```

在 WriteLine 方法的最后一行，使用变量 s2 的值把 World 字符串插值到新的占位符 0 中：

```
WriteLine("Hello, World");
```

## 9.2.2 日期时间和数字的格式

除了给占位符使用字符串格式之外，还可以根据数据类型使用特定的格式。下面先从日期开始。在占位符中，格式字符串跟在表达式的后面，用冒号隔开。下面所示的例子是用于 DateTime 类型的 D 和 d 格式：

```
var day = new DateTime(2025, 2, 14);
WriteLine($"{day:D}");
WriteLine($"{day:d}");
```

结果显示，用大写字母 D 表示长日期格式字符串，用小写字母 d 表示短日期字符串：

```
Friday, February 14, 2025
2/14/2025
```

根据所使用的大写或小写字符串，DateTime 类型会得到不同的结果。根据系统的语言设置，输出可能不同。日期和时间是特定于语言的。

DateTime 类型支持很多不同的标准格式字符串，显示日期和时间的所有表示：例如，t 表示短时间格式，T 表示长时间格式，g 和 G 显示日期和时间。这里不讨论所有其他选项，在 MSDN 文档的 DateTime 类型的 ToString 方法中，可以找到相关介绍。

**注意：**

应该提到的一个问题是，为 DateTime 构建自定义的格式字符串。自定义的日期和时间格式字符串可以结合格式说明符，例如 dd-MMM-yyyy:

```
Console.WriteLine($"{day:dd-MMM-yyyy}");
```

结果如下：

```
14-Feb-2025
```

这个自定义格式字符串利用 dd 把日期显示为两个数字(如果某个日期在 10 日之前，这就很重要，从这里可以看到 d 和 dd 之间的区别)、MMM(月份的缩写名称，注意它是大写，而 mm 表示分钟)和表示四位数年份的 yyyy。同样，在 MSDN 文档中可以找到用于自定义日期和时间格式字符串的所有其他格式说明符。

数字的格式字符串不区分大小写。下面看看 n、e、 x 和 c 标准数字格式字符串：

```
int i = 2477;
Console.WriteLine($"{i:n} {i:e} {i:x} {i:c}");
```

n 格式字符串定义了一个数字格式，用组分隔符显示整数和小数。e 表示使用指数表示法，x 表示转换为十六进制，c 显示货币：

```
2,477.00 2.477000e+003 9ad $2,477.00
```

对于数字的表示，还可以使用定制的格式字符串。#格式说明符是一个数字占位符，如果数字可用，就显示数字；如果数字不可用，就不显示数字。0 格式说明符是一个零占位符，显示相应的数字，如果数字不存在，就显示零。

```
double d = 3.1415;
Console.WriteLine($"{d:###.###}");
Console.WriteLine($"{d:000.000}");
```

在示例代码中，对于 double 值，第一个结果把逗号后的值舍入为三位小数，第二个结果是显示逗号前的三个数字：

```
3.142
003.142
```

MSDN 文档给百分比、往返和定点显示提供了所有的标准数字格式字符串，以及提供自定义格式字符串，用于使指数、小数点、组分隔符等显示不同的外观。

### 9.2.3　自定义字符串格式

格式字符串不限于内置类型，可以为自己的类型创建自定义格式字符串。为此，只需要实现接口 IFormattable。

首先是一个简单的 Person 类，它包含 FirstName 和 LastName 属性(代码文件 StringFormats/Person.cs)：

```
public class Person
{
  public string FirstName { get; set; }
  public string LastName { get; set; }
}
```

为了获得这个类的简单字符串表示，重写基类的 ToString 方法。这个方法返回由 FirstName 和 LastName 组成的字符串：

```
public override string ToString() => FirstName + " " + LastName;
```

除了简单的字符串表示之外，Person 类也应该支持格式字符串 F，返回名 L 和姓 A，后者代表“all”；并且应该提供与 ToString 方法相同的字符串表示。为实现自定义字符串，接口 IFormattable 定义了带两个参数的 ToString 方法：一个是格式的字符串参数，另一个是 IFormatProvider 参数。IFormatProvider 参数未在示例代码中使用。可以基于区域值使用这个参数，进行不同的显示，因为 CultureInfo 类实现了该接口。

实现了这个接口的其他类是 NumberFormatInfo 和 DateTimeFormatInfo。可以把实例传递到 ToString 方法的第二个参数，使用这些类配置数字和 DateTime 的字符串表示。ToString 方法的实现代码只使用 switch 语句，基于格式字符串返回不同的字符串。为了使用格式字符串直接调用 ToString 方法，而不提供格式提供程序，应重载 ToString 方法。这个方法又调用有两个参数的 ToString 方法：

```
public class Person : IFormattable
{
  public string FirstName { get; set; }
  public string LastName { get; set; }
  public override string ToString() => FirstName + " " + LastName;
  public virtual string ToString(string format) => ToString(format, null);

  public string ToString(string format, IFormatProvider formatProvider)
  {
    switch (format)
    {
      case null:
      case "A":
        return ToString();
      case "F":
        return FirstName;
      case "L":
        return LastName;
      default:
        throw new FormatException($"invalid format string {format}");
    }
  }
}
```

有了这些代码，就可以显式传递格式字符串，或隐式使用字符串插值，以调用 ToString 方法。隐式的调用使用带两个参数的 ToString 方法，并给 IFormatProvider 参数传递 null(代码文件 StringFormats/Program.cs)：

```
var p1 = new Person { FirstName = "Stephanie", LastName = "Nagel" };
Console.WriteLine(p1.ToString("F"));
Console.WriteLine($"{p1:F}");
```

## 9.3 正则表达式

正则表达式作为小型技术领域的一部分，在各种程序中都有着难以置信的作用。正则表达式可以看成一种有特定功能的小型编程语言：在大的字符串表达式中定位子字符串。它不是一种新技术，最初是在 UNIX 环境中开发的，与 Perl 和 JavaScript 编程语言一起使用得比较多。System.Text.RegularExpressions 名称空间中的许多.NET 类都支持正则表达式。.NET Framework 的各个部分也使用正则表达式。例如，在 ASP.NET 验证服务器的控件中就使用了正则表达式。

对于不太熟悉正则表达式语言的读者，本节将主要解释正则表达式和相关的.NET 类。如果你很熟悉正则表达式，就可以浏览本节，选择学习与.NET 基类引用有关的内容。注意，.NET 正则表达式引擎用于兼容 Perl 5 的正则表达式，但它有一些新功能。

### 9.3.1 正则表达式概述

正则表达式语言是一种专门用于字符串处理的语言。它包含两个功能：

- 一组用于标识特殊字符类型的转义代码。你可能很熟悉 DOS 命令中使用*字符表示任意子字符串(例如，DOS 命令 Dir Re*会列出名称以 Re 开头的所有文件)。正则表达式使用与*类似的许多序列来表示“任意一个字符”、“一个单词的中断”和“一个可选的字符”等。
- 一个系统，在搜索操作中把子字符串和中间结果的各个部分组合起来。

使用正则表达式，可以对字符串执行许多复杂而高级的操作，例如：

- 识别(可以是标记或删除)字符串中所有重复的单词，例如，把“The computer books books”转换为“The computer books”。
- 把所有单词都转换为标题格式，例如，把“this is a Title”转换为“This Is A Title”。
- 把长于 3 个字符的所有单词都转换为标题格式，例如，把“this is a Title”转换为“This is a Title”。
- 确保句子有正确的大写形式。
- 区分 URI 的各个元素(例如，给定 http://www.wrox.com，提取出其中的协议、计算机名和文件名等)。

当然，这些都是可以在 C#中用 System.String 和 System.Text.StringBuilder 的各种方法执行的任务。但是，在一些情况下，还需要编写相当多的 C#代码。如果使用正则表达式，这些代码一般可以压缩为几行。实际上，这是实例化了一个对象 System.Text.RegularExpressions.RegEx(甚至更简单，调用静态的 RegEx()方法)，给它传递要处理的字符串和一个正则表达式(这是一个字符串，它包含用正则表达式语言编写的指令)。

正则表达式字符串初看起来像是一般的字符串，但其中包含了转义序列和有特定含义的其他字符。例如，序列\b 表示一个字的开头和结尾(字的边界)，因此如果要表示正在查找以字符 th 开头的字，就可以编写正则表达式\bth(即字边界是序列-t-h)。如果要搜索所有以 th 结尾的单词，就可以编写 th\b(字边界是序列 t- h-)。但是，正则表达式要比这复杂得多，包括可以在搜索操作中找到存储部分文本的工具性程序。本节仅简要介绍正则表达式的功能。

**注意：**

正则表达式的更多信息可参阅 Andrew Watt 撰写的图书 *Beginning Regular Expressions*(John Wiley & Sons, 2005)。

假定应用程序需要把美国电话号码转换为国际格式。在美国，电话号码的格式为 314-123-1234，常常写作(314)123-1234。在把这个国家格式转换为国际格式时，必须在电话号码的前面加上+1(美国的国家代码)，并给区号加上圆括号：+1(314) 123-1234。在查找和替换时，这并不复杂。但如果要使用 String 类完成这个转换，就需要编写一些代码(这表示必须使用 System.String 类的方法来编写代码)。而正则表达式语言可以构造一个短的字符串来表达上述含义。

所以，本节只有一个非常简单的示例，我们只考虑如何查找字符串中的某些子字符串，不需要考虑如何修改它们。

## 9.3.2　RegularExpressionsPlayground 示例

本章的正则表达式示例使用如下名称空间：

System

System.Text.RegularExpressions

下面将开发一个小示例 RegularExpressionsPlayground，通过实现并显示一些搜索的结果，说明正则表达式的一些功能，以及如何在 C#中使用.NET 正则表达式引擎。在这个示例文档中使用的文本是本书前一版的部分简介(代码文件 RegularExpressionsPlayground/Program.cs)：

```
const string text =
  @"Professional C# 6 and .NET Core 1.0 provides complete coverage " +
  "of the latest updates, features, and capabilities, giving you " +
  "everything you need for C#. Get expert instruction on the latest " +
  "changes to Visual Studio 2015, Windows Runtime, ADO.NET, ASP.NET, " +
  "Windows Store Apps, Windows Workflow Foundation, and more, with " +
  "clear explanations, no-nonsense pacing, and valuable expert insight. " +
  "This incredibly useful guide serves as both tutorial and desk " +
  "reference, providing a professional-level review of C# architecture " +
  "and its application in a number of areas. You'll gain a solid " +
  "background in managed code and .NET constructs within the context of " +
  "the 2015 release, so you can get acclimated quickly and get back to work.";
```

**注意：**
上面的代码说明了前缀为@符号的逐字字符串的实用性。这个前缀在正则表达式中非常有用。

我们把这个文本称为输入字符串。为了说明.NET 类的正则表达式，我们先进行一次纯文本的基本搜索，这次搜索不带任何转义序列或正则表达式命令。假定要查找所有的字符串 ion，把这个搜索字符串称为模式。使用正则表达式和前面声明的变量 input，可编写出下面的代码(代码文件 RegularExpressionPlayground/Program.cs)：

```
public static void Find1(text)
{
  const string pattern = "ion";
  MatchCollection matches = Regex.Matches(text, pattern,
    RegexOptions.IgnoreCase | RegexOptions.ExplicitCapture);
  WriteMatches(text, matches);
}
```

在这段代码中，使用了 System.Text.RegularExpressions 名称空间中 Regex 类的静态方法 Matches()。这个方法的参数是一些输入文本、一个模式和从 RegexOptions 枚举中提取的一组可选标志。在本例中，指定所有的搜索都不应区分大小写。另一个标记 ExplicitCapture 改变了收集匹配的方式，对于本例，这样可以使搜索的效率更高，其原因详见后面的内容(尽管它还有这里没有探讨的其他用法)。Matches()方法返回 MatchCollections 对象的引用。匹配是一个技术术语，表示在表达式中查找模式实例的结果，用 System.Text.RegularExpressions.Match 类表示它。因此，我们返回一个包含所有匹配的 MatchCollection，每个匹配都用一个 Match 对象来表示。在上面的代码中，只是迭代集合，并使用 Match 类的 Index 属性，Match 类返回输入文本中匹配所在的索引。

Find1 方法的结果列出了 6 个匹配：

```
No. of matches: 6
Index: 7,       String: ion,   ofessional C#
Index: 172,     String: ion,   truction on t
Index: 300,     String: ion,   undation, and
Index: 334,     String: ion,   lanations, no
Index: 481,     String: ion,   ofessional-le
Index: 535,     String: ion,   lication in a
```

表 9-3 描述了 RegexOptions 枚举的一些成员。

表 9-3

| 成 员 名 | 说 明 |
|---|---|
| CultureInvariant | 指定忽略字符串的区域值 |
| ExplicitCapture | 修改收集匹配的方式，方法是确保把显式指定的匹配作为有效的搜索结果 |
| IgnoreCase | 忽略输入字符串的大小写 |
| IgnorePatternWhitespace | 在字符串中删除未转义的空白，启用通过#符号指定的注释 |
| Multiline | 修改字符^和$，把它们应用于每一行的开头和结尾，而不仅仅应用于整个字符串的开头和结尾 |
| RightToLeft | 从右到左地读取输入字符串，而不是默认地从左到右读取(适合于一些亚洲语言或其他以这种方式读取的语言) |
| Singleline | 指定句点的含义(.)，它原来表示单行模式，现在改为匹配每个字符 |

到目前为止，在前面的示例中，除了一些新的.NET 基类外，其他都不是新的内容。但正则表达式的能力主要取决于模式字符串，原因是模式字符串不必仅包含纯文本。如前所述，它还可以包含元字符和转义序列，其中元字符是给出命令的特定字符，而转义序列的工作方式与 C#的转义序列相同。它们都是以反斜杠(\)开头的字符，且具有特殊的含义。

例如，假定要查找以 n 开头的字，那么可以使用转义序列\b，它表示一个字的边界(字的边界是以字母数字表中的某个字符开头，或者后面是一个空白字符或标点符号)。可以编写如下代码：

```
const string pattern = @"\bn";
MatchCollection myMatches = Regex.Matches(input, pattern,
```

```
RegexOptions.IgnoreCase |
RegexOptions.ExplicitCapture);
```

注意字符串前面的符号@。要在运行时把\b 传递给.NET 正则表达式引擎，反斜杠(\)不应被 C#编译器解释为转义序列。

如果要查找以序列 ure 结尾的字，就可以使用下面的代码：

```
const string pattern = @"ure\b";
```

如果要查找以字母 a 开头、以序列 ure 结尾的所有字(在本例中仅有一个匹配的字 architecture)，就必须在上面的代码中添加一些内容。显然，我们需要一个以\ba 开头、以 ure\b 结尾的模式，但中间的内容怎么办？需要告诉应用程序，在 a 和 ure 中间的内容可以是任意长度的字符，只要这些字符不是空白即可。实际上，正确的模式如下所示。

```
const string pattern = @"\ba\S*ure\b";
```

使用正则表达式要习惯的一点是，对像这样怪异的字符序列应见怪不怪。但这个序列的工作是非常逻辑化的。转义序列\S 表示任何不是空白字符的字符。*称为限定符，其含义是前面的字符可以重复任意次，包括 0 次。序列\S*表示任意数量不是空白字符的字符。因此，上面的模式匹配以 a 开头以 ure 结尾的任何单个单词。

表 9-4 是可以使用的一些主要的特定字符或转义序列，但这个表并不完整，完整的列表请参考 MSDN 文档。

表 9-4

| 符 号 | 含　义 | 示 例 | 匹配的示例 |
|---|---|---|---|
| ^ | 输入文本的开头 | ^B | B，但只能是文本中的第一个字符 |
| $ | 输入文本的结尾 | X$ | X，但只能是文本中的最后一个字符 |
| . | 除了换行符(\)以外的所有单个字符 | i.ation | isation、ization |
| * | 可以重复 0 次或多次的前导字符 | ra*t | rt、rat、raat 和 raaat 等 |
| + | 可以重复 1 次或多次的前导字符 | ra+t | rat、raat 和 raaat 等(但不能是 rt) |
| ? | 可以重复 0 次或 1 次的前导字符 | ra?t | 只有 rt 和 rat 匹配 |
| \s | 任何空白字符 | \sa | [space]a、\ta、\na (\t 和\n 与 C#中的\t 和\n 含义相同) |
| \S | 任何不是空白的字符 | \SF | aF、rF、cF，但不能是\tf |
| \b | 字边界 | ion\b | 以 ion 结尾的任何字 |
| \B | 不是字边界的任意位置 | \BX\B | 字中间的任何 X |

如果要搜索其中一个元字符，就可以通过带有反斜杠的相应转义字符来表示。例如，“.” (一个句点)表示除了换行字符以外的任何单个字符，而 “\.” 表示一个点。

可以把替换的字符放在方括号中，请求匹配包含这些字符。例如，[1|c]表示字符可以是 1 或 c。如果要搜索 map 或 man，就可以使用序列 ma[n|p]。在方括号中，也可以指定一个范围，例如，[a-z]表示所有的小写字母，[A-E]表示 A~E 之间的所有大写字母(包括字母 A 和 E)，[0-9]表示一个数字，其简写方式是\d。如果要搜索一个整数(该序列只包含 0~9 的字符)，就可以编写[0-9]+或[\d]+。

**注意：**
使用 “+” 字符表示至少要有这样一个数字，但可以有多个数字，所以 9、 83 和 854 等都是匹配的。

^用在方括号中时有不同的含义。在方括号外部使用它，就标记输入文本的开头。在方括号内使用它，表示除了^之后的字符之外的任意字符。

## 9.3.3　显示结果

本节编写一个示例 RegularExpressionsPlayground，看看正则表达式的工作方式。

该示例的核心是方法 WriteMatches()，它把 MatchCollection 中的所有匹配以比较详细的格式显示出来。对

于每个匹配结果，该方法都会显示匹配在输入字符串中的索引、匹配的字符串和一个略长的字符串，其中包含匹配结果和输入文本中至多 10 个外围字符，其中至多有 5 个字符放在匹配结果的前面，至多 5 个字符放在匹配结果的后面(如果匹配结果的位置在输入文本的开头或结尾 5 个字符内，则结果中匹配字符串前后的字符就会少于 5 个)。换言之，在 RegularExpressionsPlayground 示例开始时，如果要匹配的单词是 applications，靠近输入文本开头的匹配结果应是"web applications imme"，匹配结果的前后各有 5 个字符，但位于输入文本的最后一个字 immediately 上的匹配结果就应是" ions immediately " ——匹配结果的后面只有一个字符，因为在该字符的后面是字符串的结尾。下面这个长字符串可以更清楚地表明正则表达式是在什么地方查找到匹配结果的(代码文件 RegularExpressionPlayground/Program.cs)：

```
public static void WriteMatches(string text, MatchCollection matches)
{
  Console.WriteLine($"Original text was: \n\n{text}\n");
  Console.WriteLine($"No. of matches: {matches.Count}");
  foreach (Match nextMatch in matches)
  {
    int index = nextMatch.Index;
    string result = nextMatch.ToString();
    int charsBefore = (index < 5) ? index : 5;
    int fromEnd = text.Length - index - result.Length;
    int charsAfter = (fromEnd < 5) ? fromEnd : 5;
    int charsToDisplay = charsBefore + charsAfter + result.Length;
    Console.WriteLine($"Index: {index}, \tString: {result}, \t" +
      "{text.Substring(index - charsBefore, charsToDisplay)}");
  }
}
```

在这个方法中，处理过程是确定在较长的子字符串中有多少个字符可以显示，而不需要超出输入文本的开头或结尾。注意在 Match 对象上使用了另一个属性 Value，它包含标识该匹配的字符串。而且，RegularExpressionsPlayground 只包含名为 Find1、Find2 等的方法，这些方法根据本节中的示例执行某些搜索操作。例如，Find2 查找以 a 开头、以 ure 结尾的任意字符串：

```
public static void Find2(string text)
{
  string pattern = @"\ba\S*ure\b";
  MatchCollection matches = Regex.Matches(text, pattern,
  RegexOptions.IgnoreCase);
  Console.WriteMatches(text, matches);
}
```

下面是一个简单的 Main()方法，可以编辑它，从而选择一个 Find<n>()方法：

```
public static void Main()
{
  Find2();
  Console.ReadLine();
}
```

这段代码还需要使用 RegularExpressions 名称空间：

```
using System;
using System.Text.RegularExpressions;
```

运行带有 Find 2()方法的示例，得到如下所示的结果：

```
No. of matches: 1
Index: 506,    String: architecture,   f C# architecture and
```

### 9.3.4 匹配、组和捕获

正则表达式的一个优秀特性是可以把字符组合起来，其工作方式与 C#中的复合语句一样。在 C#中，可以把任意数量的语句放在花括号中，把它们组合在一起，其结果视为复合语句。在正则表达式模式中，也可以把任何字符组合起来(包括元字符和转义序列)，像处理单个字符那样处理它们。唯一的区别是要使用圆括号而不是花括号，得到的序列称为一组。

例如，模式(an)+定位任意重复出现的序列 an。限定符"+"只应用于它前面的一个字符，但因为我们把字符组合起来了，所以它现在把重复的 an 作为一个单元来对待。这意味着，如果(an)+应用到输入文本"bananas came

to Europe late in the annals of history”上，就会从 bananas 中识别出 anan。另一方面，如果使用 an+，则程序将从 annals 中选择 ann，从 bananas 中选择出两个分开的 an 序列。表达式(an)+可以识别出 an、anan、ananan 等，而表达式 an+可以识别出 an、ann、annn 等。

**注意：**

在上面的示例中，为什么(an)+从 banana 中选择的是 anan，而没有把其中一个 an 作为一个匹配结果？因为匹配结果是不能重叠的。如果有可能重叠，在默认情况下就选择最长的匹配序列。

但是，组的功能要比这强大得多。在默认情况下，把模式的一部分组合为一个组时，就要求正则表达式引擎按照该组来匹配，或按照整个模式来匹配。换言之，可以把组当成一个要匹配和返回的模式。如果要把字符串分解为各个部分，这种模式就非常有效。

例如，URI 的格式是<protocol>://<address>:<port>，其中端口是可选的。它的一个示例是 http://www.wrox.com:80。假定要从一个 URI 中提取协议、地址和端口，而且不考虑 URI 的后面是否紧跟着空白(但没有标点符号)，那么可以使用下面的表达式：

```
\b(https?)(://)([.\w]+)([\s:]([\d]{2,5})?)\b
```

该表达式的工作方式如下：首先，前导\b 序列和结尾\b 序列确保只需要考虑完全是字的文本部分。在这个文本部分中，第一组(https?)会识别 http 或 https 协议。S 字符后面的?指定这个字符可能出现 0 次或 1 次，因此找到 http 和 https。括号表示把协议存储为一组。

第二组是一个简单的(://)。它仅指定字符://。

第三组([.\w]+)比较有趣。这个组包含一个放在括号里的表达式，该表达式要么是句点字符(.)，要么是用\w 指定的任意字母数字字符。这些字符可以重复任意多次，因此匹配 www.wrox.com。

第四组([\s:]([\d]{2,5})?)是一个较长的表达式，包含一个内部组。在该组中，第一个放在括号中的表达式允许通过\s 指定空白字符或冒号。内部组用[\d]指定一个数字。表达式{ 2, 5 }指定前面的字符(数字)允许至少出现两次但不超过 5 次。数字的完整表达式用内部组后面的?指定允许出现 0 次或 1 次。使这个组变成可选非常重要，因为端口号并不总是在 URI 中指定；事实上，通常不指定它。

下面定义一个字符串来运行这个表达式(代码文件 RegularExpressionsPlayground/Program.cs)：

```
string line = "Hey, I've just found this amazing URI at " +
"http:// what was it -oh yes https://www.wrox.com or " +
"http://www.wrox.com:80";
```

与这个表达式匹配的代码使用类似于之前的 Matches 方法。区别是在 Match.Groups 属性内迭代所有的 Group 对象，在控制台上输出每组得到的索引和值：

```
string pattern = @"\b(https?)(://)([.\w]+)([\s:]([\d]{2,4})?)\b";
var r = new Regex(pattern);
MatchCollection mc = r.Matches(line);
foreach (Match m in mc)
{
  Console.WriteLine($"Match: {m}");
  foreach (Group g in m.Groups)
  {
    if (g.Success)
    {
      Console.WriteLine($"group index: {g.Index}, value: {g.Value}");
    }
  }
  Console.WriteLine();
}
```

运行程序，得到如下组和值：

```
Match https://www.wrox.com
group index 70, value: https://www.wrox.com
group index 70, value: https
group index 75, value: ://
group index 78, value: www.wrox.com
group index 90, value:
Match http://www.wrox.com:80
```

```
group index 94, value http://www.wrox.com:80
group index 94, value: http
group index 98, value: ://
group index 101, value: www.wrox.com
group index 113, value: :80
group index 114, value: 80
```

之后，就匹配文本中的 URI，URI 的不同部分得到了很好的分组。组还提供了更多的功能。一些组，如协议和地址之间的分隔，可以忽略，并且组也可以命名。

修改正则表达式，命名每个组，忽略一些名称。在组的开头指定?<name>，就可给组命名。例如，协议、地址和端口的正则表达式组就采用相应的名称。在组的开头使用?:来忽略该组。不要迷惑于组内的?::/ /，它表示搜索:/ /，组本身因为前面的?:而被忽略：

```
string pattern = @"\b(?<protocol>https?)(?:://)" +
@"(?<address>[.\w]+)([\s:](?<port>[\d]{2,4})?)\b";
```

为了从正则表达式中获得组，Regex 类定义了 GetGroupNames 方法。在下面的代码段中，每个匹配都使用所有的组名，使用 Groups 属性和索引器输出组名和值：

```
Regex r = new Regex(pattern, RegexOptions.ExplicitCapture);
MatchCollection mc = r.Matches(line);
foreach (Match m in mc)
{
  Console.WriteLine($"match: {m} at {m.Index}");
  foreach (var groupName in r.GetGroupNames())
  {
    Console.WriteLine($"match for {groupName}: {m.Groups[groupName].Value}");
  }
}
```

运行程序，就可以看到组名及其值：

```
match: https://www.wrox.com at 70
match for 0: https://www.wrox.com
match for protocol: https
match for address: www.wrox.com
match for port:
match: http://www.wrox.com:80 at 94
match for 0: http://www.wrox.com:80
match for protocol: http
match for address: www.wrox.com
match for port: 80
```

## 9.4 字符串和 Span

今天的编程代码通常处理需要操作的长字符串。例如，Web API 以 JSON 或 XML 格式返回一个长字符串。将如此大的字符串分割成许多更小的字符串，意味着创建了许多对象，而垃圾收集器不再需要这些字符串时，需要做很多事情来释放这些字符串所占的内存。

.NET Core 有一个新的方法：Span<T>类型。这一类型参阅第 7 章。该类型引用数组的一个切片，而不需要复制它的内容。同样，Span<T>可以用来引用一个字符串的片段，而不需要复制原始内容。

下面的代码片段从一个由变量文本引用的非常长的字符串中创建了 span。它与以前使用正则表达式的字符串相同。ReadOnlySpan<char>从 AsSpan 扩展方法中返回。AsSpan 扩展了字符串类型，并返回一个 ReadOnlySpan<char>，因为字符串由 char 元素组成。在内部，Span<T>使用 ref 关键字来保存引用。在 Slice 方法中，可以从完整的字符串中获取一个切片。用第一个参数选择开头，在该索引中，在字符串中找到的第一个文本 Visual 是索引。从那里，第二个参数使用 13 个字符定义。结果还是一个 ReadOnlySpan。只有使用 span 的 ToArray 方法才能分配内存。ToArray 方法分配切片所需的内存。然后将 char 数组传递给 string 类型的构造函数，以创建一个新的字符串(代码文件 SpanWithString /Program.cs)：

```
int ix = text.IndexOf("Visual");
ReadOnlySpan<char> spanToText = text.AsSpan();
ReadOnlySpan<char> slice = spanToText.Slice(ix, 13);

string newString = new string(slice.ToArray());
```

```
Console.WriteLine(newString);
```

从切片中新分配的字符串包含 Visual Studio。

**注意：**

第 7 章介绍了 Span 和数组。第 17 章讨论了 Span 映射到本机内存和 ref 关键字。返回 JSON 或 XML 的 Web API 在第 32 章中介绍。JSON 和 XML 的详细信息可以参阅网上附加第 2 章。

## 9.5　小结

在使用.NET Framework 时，可用的数据类型相当多。在应用程序(特别是关注数据提交和检索的应用程序)中，最常用的一种类型就是 String 数据类型。String 非常重要，这也是本书用一整章的篇幅介绍如何在应用程序中使用和处理 String 数据类型的原因。

过去在使用字符串时，常常需要通过连接来分解字符串。而在.NET Framework 中，可以使用 StringBuilder 类完成许多这类任务，而且性能更好。

字符串的另一个特点是字符串插值。在大多数应用程序中，该特性使字符串的处理容易得多。

最后，使用正则表达式进行高级的字符串处理是搜索和验证字符串的一种最佳工具。

接下来的两章介绍不同的集合类。

# 第 10 章

# 集　　合

**本章要点**

- 理解集合接口和类型
- 使用列表、队列和栈
- 使用链表和有序列表
- 使用字典和集
- 评估性能

**本章源代码下载地址(wrox.com)：**

打开 www.wrox.com 的 Download Code 选项卡可下载本章源代码。源代码也可以在 Delegates 目录的 https://github.com/ProfessionalCSharp/ProfessionalCSharp7 中找到。本章代码分为以下几个主要的示例文件：

- 列表示例(List Samples)
- 队列示例(Queue Sample)
- 链表示例(Linked List Sample)
- 有序列表示例(Sorted List Sample)
- 字典示例(Dictionary Sample)
- 集示例(Set Sample)

## 10.1　概述

第 7 章介绍了数组和 Array 类实现的接口。数组的大小是固定的。如果元素个数是动态的，就应使用集合类。

List<T>是与数组相当的集合类。还有其他类型的集合：队列、栈、链表、字典和集。其他集合类提供的访问集合元素的 API 可能稍有不同，它们在内存中存储元素的内部结构也有区别。本章将介绍所有的集合类和它们的区别，包括性能差异。

# 10.2 集合接口和类型

大多数集合类都可在 System.Collections 和 System.Collections.Generic 名称空间中找到。泛型集合类位于 System.Collections.Generic 名称空间中；专用于特定类型的集合类位于 System.Collections. Specialized 名称空间中。线程安全的集合类位于 System.Collections.Concurrent 名称空间中。不可变的集合类在 System.Collections.Immutable 名称空间中。

当然，组合集合类还有其他方式。集合可以根据集合类实现的接口组合为列表、集合和字典。

**注意：**
接口 IEnumerable 和 IEnumerator 的内容详见第 7 章。

集合和列表实现的接口如表 10-1 所示。

表 10-1

| 接 口 | 说 明 |
|---|---|
| IEnumerable<T> | 如果将 foreach 语句用于集合，就需要 IEnumerable 接口。这个接口定义了方法 GetEnumerator()，它返回一个实现了 IEnumerator 接口的枚举 |
| ICollection<T> | ICollection<T>接口由泛型集合类实现。使用这个接口可以获得集合中的元素个数(Count 属性)，把集合复制到数组中(CopyTo()方法)，还可以从集合中添加和删除元素(Add()、Remove()、Clear()) |
| IList<T> | IList<T>接口用于可通过位置访问其中的元素列表，这个接口定义了一个索引器，可以在集合的指定位置插入或删除某些项(Insert()和 RemoveAt()方法)。IList<T>接口派生自 ICollection<T>接口 |
| ISet<T> | ISet<T>接口由集实现。集允许合并不同的集，获得两个集的交集，检查两个集是否重叠。ISet<T>接口派生自 ICollection<T>接口 |
| IDictionary<TKey, TValue> | IDictionary<TKey,TValue>接口由包含键和值的泛型集合类实现。使用这个接口可以访问所有的键和值，使用键类型的索引器可以访问某些项，还可以添加或删除某些项 |
| ILookup<TKey, TValue> | ILookup<TKey，TValue>接口类似于 IDictionary<TKey,TValue>接口，实现该接口的集合有键和值，且可以通过一个键包含多个值 |
| IComparer<T> | 接口 IComparer<T>由比较器实现，通过 Compare()方法给集合中的元素排序 |
| IEqualityComparer<T> | 接口 IEqualityComparer<T>由一个比较器实现，该比较器可用于字典中的键。使用这个接口，可以对对象进行相等性比较 |

# 10.3 列表

.NET Framework 为动态列表提供了泛型类 List<T>。这个类实现了 IList、ICollection、IEnumerable、IList<T>、ICollection<T>和 IEnumerable<T>接口。

下面的例子将 Racer 类中的成员用作要添加到集合中的元素，以表示一级方程式的一位赛车手。这个类有 5 个属性：Id、Firstname、Lastname、Country 和 Wins 的次数。在该类的构造函数中，可以传递赛车手的姓名和获胜次数，以设置成员。重写 ToString()方法是为了返回赛车手的姓名。Racer 类也实现了泛型接口 IComparable<T>，为 Racer 类中的元素排序，还实现了 IFormattable 接口(代码文件 ListSamples/Racer.cs)。

```
public class Racer: IComparable<Racer>, IFormattable
{
  public int Id { get; }
  public string FirstName { get; }
  public string LastName { get; }
  public string Country { get; }
  public int Wins { get; }
```

```
    public Racer(int id, string firstName, string lastName, string country)
      :this(id, firstName, lastName, country, wins: 0)
    { }

    public Racer(int id, string firstName, string lastName, string country,int wins)
    {
      Id = id;
      FirstName = firstName;
      LastName = lastName;
      Country = country;
      Wins = wins;
    }

    public override string ToString() => $"{FirstName} {LastName}";

    public string ToString(string format, IFormatProvider formatProvider)
    {
      if (format == null) format = "N";
      switch (format.ToUpper())
      {
        case "N": // name
          return ToString();
        case "F": // first name
          return FirstName;
        case "L": // last name
          return LastName;
        case "W": // Wins
          return $"{ToString()}, Wins: {Wins}";
        case "C": // Country
          return $"{ToString()}, Country: {Country}";
        case "A": // All
          return $"{ToString()}, Country: {Country} Wins: {Wins}";
        default:
          throw new FormatException(String.Format(formatProvider,
            $"Format {format} is not supported"));
      }
    }

    public string ToString(string format) => ToString(format, null);

    public int CompareTo(Racer other)
    {
      int compare = LastName?.CompareTo(other?.LastName) ?? -1;
      if (compare == 0)
      {
        return FirstName?.CompareTo(other?.FirstName) ?? -1;
      }
      return compare;
    }
  }
```

### 10.3.1 创建列表

调用默认的构造函数，就可以创建列表对象。在泛型类 List<T>中，必须为声明为列表的值指定类型。下面的代码说明了如何声明一个包含 int 的 List<T>泛型类和一个包含 Racer 元素的列表。ArrayList 是一个非泛型列表，它可以将任意 Object 类型作为其元素。

使用默认的构造函数创建一个空列表。元素添加到列表中后，列表的容量就会扩大为可接纳 4 个元素。如果添加了第 5 个元素，列表的大小就重新设置为包含 8 个元素。如果 8 个元素还不够，列表的大小就重新设置为包含 16 个元素。每次都会将列表的容量重新设置为原来的 2 倍。

```
var intList = new List<int>();
var racers = new List<Racer>();
```

如果列表的容量改变了，整个集合就要重新分配到一个新的内存块中。在 List<T>泛型类的实现代码中，使用了一个 T 类型的数组。通过重新分配内存，创建一个新数组，Array.Copy()方法将旧数组中的元素复制到新数组中。为节省时间，如果事先知道列表中元素的个数，就可以用构造函数定义其容量。下面创建了一个容量为 10 个元素的集合。如果该容量不足以容纳要添加的元素，就把集合的大小重新设置为包含 20 或 40 个元素，每次都是原来的 2 倍。

```
List<int> intList = new List<int>(10);
```

使用 Capacity 属性可以获取和设置集合的容量。

```
intList.Capacity = 20;
```

容量与集合中元素的个数不同。集合中的元素个数可以用 Count 属性读取。当然，容量总是大于或等于元素个数。只要不把元素添加到列表中，元素个数就是 0。

```
Console.WriteLine(intList.Count);
```

如果已经将元素添加到列表中，且不希望添加更多的元素，就可以调用 TrimExcess()方法，去除不需要的容量。但是，因为重新定位需要时间，所以如果元素个数超过了容量的 90%，TrimExcess()方法就什么也不做。

```
intList.TrimExcess();
```

### 1. 集合初始值设定项

还可以使用集合初始值设定项给集合赋值。集合初始化器的语法类似于数组初始化器（参见第 7 章）。使用集合初始值设定项，可以在初始化集合时，在花括号中给集合赋值：

```
var intList = new List<int>() {1, 2};
var stringList = new List<string>() { "one", "two" };
```

**注意：**

*集合初始值设定项没有反映在已编译的程序集的 IL 代码中。编译器会把集合初始值设定项转换成对初始值设定项列表中的每一项调用 Add()方法。*

### 2. 添加元素

使用 Add()方法可以给列表添加元素，如下所示。实例化的泛型类型定义了 Add()方法的参数类型：

```
var intList = new List<int>();
intList.Add(1);
intList.Add(2);
var stringList = new List<string>();
stringList.Add("one");
stringList.Add("two");
```

把 racers 变量定义为 List<Racer>类型。使用 new 运算符创建相同类型的一个新对象。因为类 List<T>用具体类 Racer 来实例化，所以现在只有 Racer 对象可以用 Add()方法添加。在下面的示例代码中，创建了 5 个一级方程式赛车手，并把它们添加到集合中。前 3 个用集合初始值设定项添加，后两个通过显式调用 Add()方法来添加(代码文件 ListSamples/Program.cs)。

```
var graham = new Racer(7, "Graham", "Hill", "UK", 14);
var emerson = new Racer(13, "Emerson", "Fittipaldi", "Brazil", 14);
var mario = new Racer(16, "Mario", "Andretti", "USA", 12);
var racers = new List<Racer>(20) {graham, emerson, mario};
racers.Add(new Racer(24, "Michael", "Schumacher", "Germany", 91));
racers.Add(new Racer(27, "Mika", "Hakkinen", "Finland", 20));
```

使用 List<T>类的 AddRange()方法，可以一次给集合添加多个元素。因为 AddRange()方法的参数是 IEnumerable<T>类型的对象，所以也可以传递一个数组，如下所示(代码文件 ListSamples/Program.cs)：

```
racers.AddRange(new Racer[] {
  new Racer(14, "Niki", "Lauda", "Austria", 25),
  new Racer(21, "Alain", "Prost", "France", 51)});
```

**注意：**

*集合初始值设定项只能在声明集合时使用。AddRange()方法则可以在初始化集合后调用。如果在创建集合后动态获取数据，就需要调用 AddRange()。*

如果在实例化列表时知道集合的元素个数，就也可以将实现 IEnumerable<T>类型的任意对象传递给类的构造函数。这非常类似于 AddRange()方法(代码文件 ListSamples/Program.cs)：

```
var racers = new List<Racer>(
  new Racer[] {
    new Racer(12, "Jochen", "Rindt", "Austria", 6),
    new Racer(22, "Ayrton", "Senna", "Brazil", 41) });
```

### 3. 插入元素

使用 Insert()方法可以在指定位置插入元素(代码文件 ListSamples/Program.cs)：

```
racers.Insert(3, new Racer(6, "Phil", "Hill", "USA", 3));
```

方法 InsertRange()提供了插入大量元素的功能，类似于前面的 AddRange()方法。

如果索引集大于集合中的元素个数，就抛出 ArgumentOutOfRangeException 类型的异常。

### 4. 访问元素

实现了 IList 和 IList<T>接口的所有类都提供了一个索引器，所以可以使用索引器，通过传递元素号来访问元素。第一个元素可以用索引值 0 来访问。指定 racers[3]，可以访问列表中的第 4 个元素：

```
Racer r1 = racers[3];
```

可以使用 Count 属性确定元素个数，再使用 for 循环遍历集合中的每个元素，并使用索引器访问每一项(代码文件 ListSamples/Program.cs)：

```
for (int i = 0; i < racers.Count; i++)
{
  Console.WriteLine(racers[i]);
}
```

> **注意：**
> 可以通过索引访问的集合类有 ArrayList、StringCollection 和 List<T>。

因为 List<T>集合类实现了 IEnumerable 接口，所以也可以使用 foreach 语句遍历集合中的元素(代码文件 ListSamples/Program.cs)。

```
foreach (var r in racers)
{
  Console.WriteLine(r);
}
```

> **注意：**
> 编译器解析 foreach 语句时，利用了 IEnumerable 和 IEnumerator 接口，参见第 7 章。

### 5. 删除元素

删除元素时，可以利用索引，也可以传递要删除的元素。下面的代码把 3 传递给 RemoveAt()方法，删除第 4 个元素：

```
racers.RemoveAt(3);
```

也可以直接将 Racer 对象传送给 Remove()方法，来删除这个元素。按索引删除比较快，因为必须在集合中搜索要删除的元素。Remove()方法先在集合中搜索，用 IndexOf()方法获取元素的索引，再使用该索引删除元素。IndexOf()方法先检查元素类型是否实现了 IEquatable<T>接口。如果是，就调用这个接口的 Equals()方法，确定集合中的元素是否等于传递给 Equals()方法的元素。如果没有实现这个接口，就使用 Object 类的 Equals()方法比较这些元素。Object 类中 Equals()方法的默认实现代码对值类型进行按位比较，对引用类型只比较其引用。

> **注意：**
> 第 8 章介绍了如何重写 Equals()方法。

这里从集合中删除了变量 graham 引用的赛车手。变量 graham 是前面在填充集合时创建的。因为 IEquatable<T>接口和 Object.Equals()方法都没有在 Racer 类中重写，所以不能用要删除元素的相同内容创建一个新对象，再把它传递给 Remove()方法(代码文件 ListSamples/Program.cs)。

```
if (!racers.Remove(graham))
{
  Console.WriteLine("object not found in collection");
}
```

RemoveRange()方法可以从集合中删除许多元素。它的第一个参数指定了开始删除的元素索引，第二个参数指定了要删除的元素个数。

```
int index = 3;
int count = 5;
racers.RemoveRange(index, count);
```

要从集合中删除有指定特性的所有元素，可以使用 RemoveAll()方法。这个方法在搜索元素时使用下面将讨论的 Predicate<T>参数。要删除集合中的所有元素，可以使用 ICollection<T>接口定义的 Clear()方法。

#### 6. 搜索

有不同的方式在集合中搜索元素。可以获得要查找的元素的索引，或者搜索元素本身。可以使用的方法有 IndexOf()、LastIndexOf()、FindIndex()、FindLastIndex()、Find()和 FindLast()。如果只检查元素是否存在，List<T>类就提供了 Exists()方法。

IndexOf()方法需要将一个对象作为参数，如果在集合中找到该元素，这个方法就返回该元素的索引。如果没有找到该元素，就返回-1。IndexOf()方法使用 IEquatable<T>接口来比较元素(代码文件 ListSamples/Program.cs)。

```
int index1 = racers.IndexOf(mario);
```

使用 IndexOf()方法，还可以指定不需要搜索整个集合，但必须指定从哪个索引开始搜索以及比较时要迭代的元素个数。

除了使用 IndexOf()方法搜索指定的元素之外，还可以搜索有某个特性的元素，该特性可以用 FindIndex()方法来定义。FindIndex()方法需要一个 Predicate 类型的参数：

```
public int FindIndex(Predicate<T> match);
```

Predicate<T>类型是一个委托，该委托返回一个布尔值，并且需要把类型 T 作为参数。如果 Predicate<T>委托返回 true，就表示有一个匹配元素，并且找到了相应的元素。如果它返回 false，就表示没有找到元素，搜索将继续。

```
public delegate bool Predicate<T>(T obj);
```

在 List<T>类中，把 Racer 对象作为类型 T，所以可以将一个方法(该方法将类型 Racer 定义为一个参数且返回一个布尔值)的地址传递给 FindIndex()方法。查找指定国家的第一个赛车手时，可以创建如下所示的 FindCountry 类。FindCountryPredicate()方法的签名和返回类型通过 Predicate<T>委托定义。Find()方法使用变量 country 搜索用 FindCountry 类的构造函数定义的某个国家(代码文件 ListSamples/FindCountry.cs)。

```
public class FindCountry
{
  public FindCountry(string country) => _country = country;

  private string _country;

  public bool FindCountryPredicate(Racer racer) =>
    racer?.Country == _country;
}
```

使用 FindIndex()方法可以创建 FindCountry 类的一个新实例，把表示一个国家的字符串传递给构造函数，再传递 Find()方法的地址。在下面的示例中，FindIndex()方法成功完成后，index2 就包含集合中赛车手的 Country 属性设置为 Finland 的第一项的索引(代码文件 ListSamples/Program.cs)。

```
int index2 = racers.FindIndex(new FindCountry("Finland").FindCountryPredicate);
```

除了用处理程序方法创建类之外，还可以在这里创建 lambda 表达式。结果与前面完全相同。现在 lambda 表达式定义了实现代码，来搜索 Country 属性设置为 Finland 的元素。

```
int index3 = racers.FindIndex(r => r.Country == "Finland");
```

与 IndexOf()方法类似，使用 FindIndex()方法也可以指定搜索开始的索引和要遍历的元素个数。为了从集合中的最后一个元素开始向前搜索某个索引，可以使用 FindLastIndex()方法。

FindIndex()方法返回所查找元素的索引。除了获得索引之外，还可以直接获得集合中的元素。Find()方法需要一个 Predicate<T>类型的参数，这与 FindIndex()方法类似。下面的 Find()方法搜索列表中 FirstName 属性设置为 Niki 的第一个赛车手。当然，也可以实现 FindLast()方法，查找与 Predicate<T>类型匹配的最后一项。

```
Racer racer = racers.Find(r => r.FirstName == "Niki");
```

要获得与 Predicate<T>类型匹配的所有项，而不是一项，可以使用 FindAll()方法。FindAll()方法使用的 Predicate<T>委托与 Find()和 FindIndex()方法相同。FindAll()方法在找到第一项后，不会停止搜索，而是继续迭代集合中的每一项，并返回 Predicate<T>类型是 true 的所有项。

这里调用了 FindAll()方法，返回 Wins 属性设置为大于 20 的整数的所有 racer 项。从 bigWinners 列表中引用所有赢得超过 20 场比赛的赛车手。

```
List<Racer> bigWinners = racers.FindAll(r => r.Wins > 20);
```

用 foreach 语句遍历 bigWinners 变量，结果如下：

```
foreach (Racer r in bigWinners)
{
  Console.WriteLine($"{r:A}");
}
Michael Schumacher, Germany Wins: 91
Niki Lauda, Austria Wins: 25
Alain Prost, France Wins: 51
```

这个结果没有排序，但这是下一步要做的工作。

**注意：**
格式修饰符和 IFormattable 接口参见第 9 章。

### 7. 排序

List<T>类可以使用 Sort()方法对元素排序。Sort()方法使用快速排序算法，比较所有的元素，直到整个列表排好序为止。

Sort()方法使用了几个重载的方法。可以传递给它的参数有泛型委托 Comparison<T>和泛型接口 IComparer<T>，以及一个范围值和泛型接口 IComparer<T>。

```
public void List<T>.Sort();
public void List<T>.Sort(Comparison<T>);
public void List<T>.Sort(IComparer<T>);
public void List<T>.Sort(Int32, Int32, IComparer<T>);
```

只有集合中的元素实现了 IComparable 接口，才能使用不带参数的 Sort()方法。

Racer 类实现了 IComparable<T>接口，可以按姓氏对赛车手排序：

```
racers.Sort();
```

如果需要按照元素类型不默认支持的方式排序，就应使用其他技术，例如，传递一个实现了 IComparer<T>接口的对象。

RacerComparer 类为 Racer 类型实现了接口 IComparer<T>。这个类允许按名字、姓氏、国籍或获胜次数排序。排序的种类用内部枚举类型 CompareType 定义。CompareType 枚举类型用 RacerComparer 类的构造函数设置。IComparer<Racer>接口定义了排序所需的 Compare()方法。在这个方法的实现代码中，使用了 string 和 int 类型的 CompareTo()方法(代码文件 ListSamples/RacerComparer.cs)。

```
public class RacerComparer : IComparer<Racer>
{
  public enum CompareType
```

```
    {
      FirstName,
      LastName,
      Country,
      Wins
    }
    private CompareType _compareType;
    public RacerComparer(CompareType compareType)
    {
      _compareType = compareType;
    }

    public int Compare(Racer x, Racer y)
    {
      if (x == null && y == null) return 0;
      if (x == null) return -1;
      if (y == null) return 1;
      int result;
      switch (_compareType)
      {
        case CompareType.FirstName:
          return string.Compare(x.FirstName, y.FirstName);
        case CompareType.LastName:
          return string.Compare(x.LastName, y.LastName);
        case CompareType.Country:
          result = string.Compare(x.Country, y.Country);
          if (result == 0)
            return string.Compare(x.LastName, y.LastName);
          else
            return result;
        case CompareType.Wins:
          return x.Wins.CompareTo(y.Wins);
        default:
          throw new ArgumentException("Invalid Compare Type");
      }
    }
  }
```

**注意：**

如果传递给 Compare 方法的两个元素的顺序相同，该方法则返回 0。如果返回值小于 0，说明第一个参数小于第二个参数；如果返回值大于 0，则第一个参数大于第二个参数。传递 null 作为参数时，Compare 方法并不会抛出一个 NullReferenceException 异常。相反，因为 null 的位置在其他任何元素之前，所以如果第一个参数为 null，该方法返回-1，如果第二个参数为 null，则返回+1。

现在，可以对 RacerComparer 类的一个实例使用 Sort()方法。传递枚举 RacerComparer. CompareType.Country，按属性 Country 对集合排序：

```
racers.Sort(new RacerComparer(RacerComparer.CompareType.Country));
```

排序的另一种方式是使用重载的 Sort()方法，该方法需要一个 Comparison<T>委托：

```
public void List<T>.Sort(Comparison<T>);
```

Comparison<T>是一个方法的委托，该方法有两个 T 类型的参数，返回类型为 int。如果参数值相等，该方法就必须返回 0。如果第一个参数比第二个小，它就必须返回一个小于 0 的值；否则，必须返回一个大于 0 的值。

```
public delegate int Comparison<T>(T x, T y);
```

现在可以把一个 lambda 表达式传递给 Sort()方法，按获胜次数排序。两个参数的类型是 Racer，在其实现代码中，使用 int 类型的 CompareTo()方法比较 Wins 属性。在实现代码中，因为以逆序方式使用 r2 和 r1，所以获胜次数以降序方式排序。调用方法之后，完整的赛车手列表就按赛车手的获胜次数排序。

```
racers.Sort((r1, r2) => r2.Wins.CompareTo(r1.Wins));
```

也可以调用 Reverse()方法，逆转整个集合的顺序。

### 10.3.2 只读集合

创建集合后，它们就是可读写的，否则就不能给它们填充值了。但是，在填充完集合后，可以创建只读集合。List<T>集合的 AsReadOnly()方法返回 ReadOnlyCollection<T>类型的对象。ReadOnlyCollection<T>类实现的接口与 List<T>集合相同，但所有修改集合的方法和属性都抛出 NotSupportedException 异常。除了 List<T>的接口之外，ReadOnlyCollection<T>还实现了 IReadOnlyCollection<T>和 IReadOnlyList<T>接口。因为这些接口的成员，集合不能修改。

## 10.4 队列

队列是其元素以先进先出(FirstIn, FirstOut, FIFO)的方式来处理的集合。先放入队列中的元素会先读取。队列的例子有在机场排的队列、人力资源部中等待处理求职信的队列和打印队列中等待处理的打印任务，以及按循环方式等待 CPU 处理的线程。另外，还常常有元素根据其优先级来处理的队列。

例如，在机场的队列中，商务舱乘客的处理要优先于经济舱的乘客。这里可以使用多个队列，一个队列对应一个优先级。在机场，这很常见，因为商务舱乘客和经济舱乘客有不同的登记队列。打印队列和线程也是这样。可以为一组队列建立一个数组，数组中的一项代表一个优先级。在每个数组项中都有一个队列，其中按照 FIFO 的方式进行处理。

> **注意：**
> 本章的后面将使用链表的另一种实现方式来定义优先级列表。

队列使用 System.Collections.Generic 名称空间中的泛型类 Queue<T>实现。在内部，Queue<T>类使用 T 类型的数组，这类似于 List<T>类型。它实现 ICollection 和 IEnumerable<T>接口，但没有实现 ICollection<T>接口，因为这个接口定义的 Add()和 Remove()方法不能用于队列。

因为 Queue<T>类没有实现 IList<T>接口，所以不能用索引器访问队列。队列只允许在队列中添加元素，该元素会放在队列的尾部(使用 Enqueue()方法)，从队列的头部获取元素(使用 Dequeue()方法)。

图 10-1 显示了队列的元素。Enqueue()方法在队列的一端添加元素，Dequeue()方法在队列的另一端读取和删除元素。再次调用 Dequeue()方法，会删除队列中的下一项。

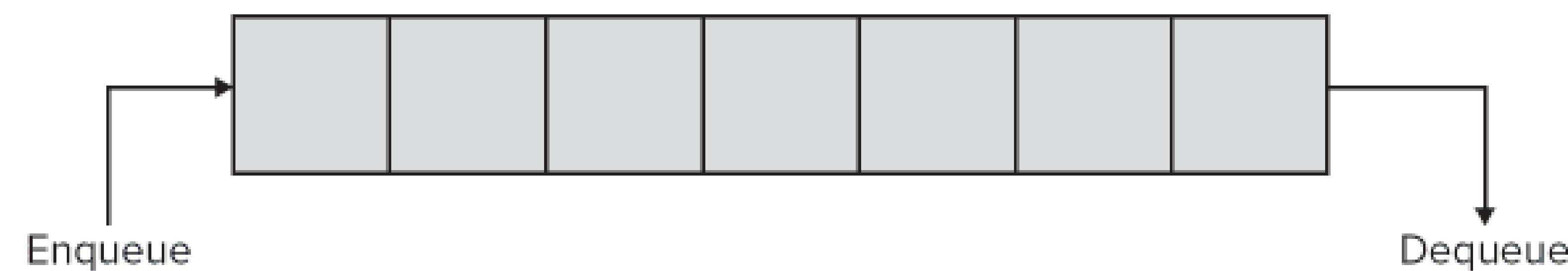



图 10-1

Queue<T>类的方法如表 10-2 所示。

表 10-2

| Queue<T>类的成员 | 说　明 |
|---|---|
| Count | Count 属性返回队列中的元素个数 |
| Enqueue | Enqueue()方法在队列一端添加一个元素 |
| Dequeue | Dequeue()方法在队列的头部读取和删除元素。如果在调用 Dequeue()方法时，队列中不再有元素，就抛出一个 InvalidOperationException 类型的异常 |
| Peek | Peek()方法从队列的头部读取一个元素，但不删除它 |
| TrimExcess | TrimExcess()方法重新设置队列的容量。Dequeue()方法从队列中删除元素，但它不会重新设置队列的容量。要从队列的头部去除空元素，应使用 TrimExcess()方法 |

在创建队列时，可以使用与 List<T>类型类似的构造函数。虽然默认的构造函数会创建一个空队列，但也可以使用构造函数指定容量。在把元素添加到队列中时，如果没有定义容量，容量就会递增，从而包含 4、8、16 和 32 个元素。类似于 List<T>类，队列的容量也总是根据需要成倍增加。非泛型类 Queue 的默认构造函数与此不同，它会创建一个包含 32 项空的初始数组。使用构造函数的重载版本，还可以将实现了 IEnumerable<T>接口的其他集合复制到队列中。

下面的文档管理应用程序示例说明了 Queue<T>类的用法。使用一个线程将文档添加到队列中，用另一个线程从队列中读取文档，并处理它们。

存储在队列中的项是 Document 类型。Document 类定义了标题和内容(代码文件 QueueSample/Document.cs)：

```
public class Document
{
  public string Title { get; }
  public string Content { get; }
  public Document(string title, string content)
  {
    Title = title;
    Content = content;
  }
}
```

DocumentManager 类是 Queue<T>类外面的一层。DocumentManager 类定义了如何处理文档：用 AddDocument()方法将文档添加到队列中，用 GetDocument()方法从队列中获得文档。

在 AddDocument()方法中，用 Enqueue()方法把文档添加到队列的尾部。在 GetDocument()方法中，用 Dequeue()方法从队列中读取第一个文档。因为多个线程可以同时访问 DocumentManager 类，所以用 lock 语句锁定对队列的访问。

> **注意：**
> 线程和 lock 语句参见第 21 章。

IsDocumentAvailable 是一个只读类型的布尔属性，如果队列中还有文档，它就返回 true，否则返回 false(代码文件 QueueSample/DocumentManager.cs)。

```
public class DocumentManager
{
  private readonly object _syncQueue = new object();
  private readonly Queue<Document> _documentQueue = new Queue<Document>();

  public void AddDocument(Document doc)
  {
    lock (_syncQueue)
    {
      _documentQueue.Enqueue(doc);
    }
  }

  public Document GetDocument()
  {
    Document doc = null;
    lock (_syncQueue)
    {
      doc = _documentQueue.Dequeue();
    }
    return doc;
  }

  public bool IsDocumentAvailable => _documentQueue.Count > 0;
}
```

ProcessDocuments 类在一个单独的任务中处理队列中的文档。能从外部访问的唯一方法是 Start()。在 Start()方法中，实例化了一个新任务。创建一个 ProcessDocuments 对象，来启动任务，定义 Run()方法作为任务的启动方法。TaskFactory(通过 Task 类的静态属性 Factory 访问)的 StartNew 方法需要一个 Action 委托作为参数，用于接受 Run 方法传递的地址。TaskFactory 的 StartNew 方法会立即启动任务。

使用 ProcessDocuments 类的 Run()方法定义一个无限循环。在这个循环中，使用属性 IsDocumentAvailable

确定队列中是否还有文档。如果队列中还有文档，就从 DocumentManager 类中提取文档并处理。这里的处理仅是把信息写入控制台。在真正的应用程序中，文档可以写入文件、数据库，或通过网络发送(代码文件 QueueSample/ProcessDocuments.cs)。

```
public class ProcessDocuments
{
  public static Task Start(DocumentManager dm) =>
    Task.Run(new ProcessDocuments(dm).Run);

  protected ProcessDocuments(DocumentManager dm) =>
    _documentManager = dm ?? throw new ArgumentNullExcption(nameof(dm));

  private DocumentManager _documentManager;

  protected async Task Run()
  {
    while (true)
    {
      if (_documentManager.IsDocumentAvailable)
      {
        Document doc = _documentManager.GetDocument();
        Console.WriteLine("Processing document {0}", doc.Title);
      }
      await Task.Delay(new Random().Next(20));
    }
  }
}
```

在应用程序的 Main()方法中，实例化一个 DocumentManager 对象，启动文档处理任务。接着创建 1000 个文档，并添加到 DocumentManager 对象中(代码文件 QueueSample/Program.cs):

```
public class Program
{
  public static async Task Main()
  {
    var dm = new DocumentManager();

    Task processDocuments = ProcessDocuments.Start(dm);

    // Create documents and add them to the DocumentManager
    for (int i = 0; i < 1000; i++)
    {
      var doc = new Document($"Doc {i.ToString()}", "content");
      dm.AddDocument(doc);
      Console.WriteLine($"Added document {doc.Title}");
      await Task.Delay(new Random().Next(20));
    }
    await processDocuments;
    Console.ReadLine();
  }
}
```

**注意：**

使用 QueueSample，可以声明 Main()方法来返回一个任务。该特性至少需要 C# 7.1。异步的 Main()方法详见第 15 章。

在启动应用程序时，会在队列中添加和删除文档，输出如下所示：

```
Added document Doc 279
Processing document Doc 236
Added document Doc 280
Processing document Doc 237
Added document Doc 281
Processing document Doc 238
Processing document Doc 239
Processing document Doc 240
Processing document Doc 241
Added document Doc 282
Processing document Doc 242
Added document Doc 283
Processing document Doc 243
```

完成示例应用程序中描述的任务的真实程序可以处理用 Web 服务接收到的文档。

## 10.5　栈

栈是与队列非常类似的另一个容器，只是要使用不同的方法访问栈。最后添加到栈中的元素会最先读取。栈是一个后进先出(LastIn, FirstOut, LIFO)的容器。

图 10-2 表示一个栈，用 Push()方法在栈中添加元素，用 Pop()方法获取最近添加的元素。

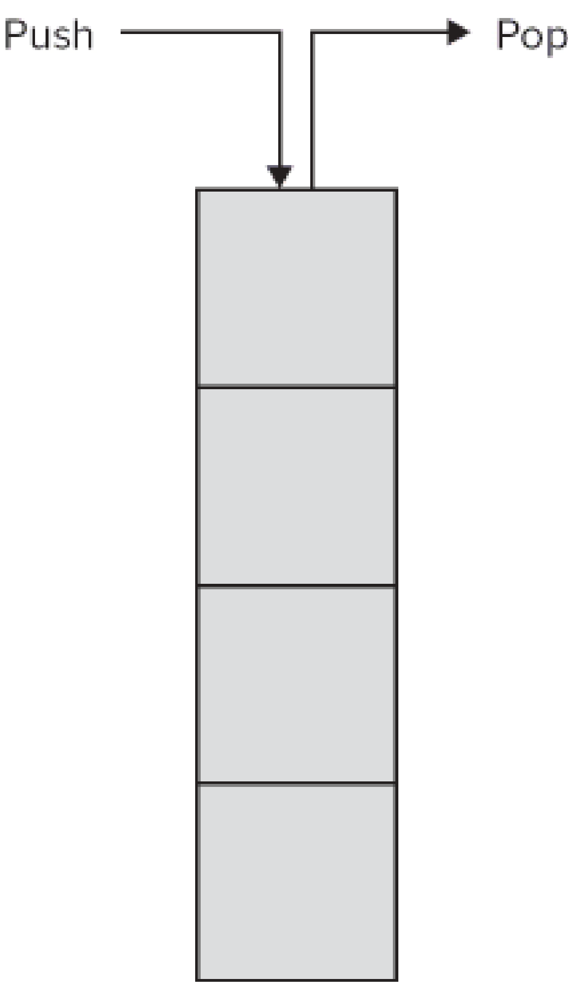



图 10-2

与 Queue<T>类相似，Stack<T>类实现 IEnumerable<T>和 ICollection 接口。

Stack<T>类的成员如表 10-3 所示。

表 10-3

| Stack<T>类的成员 | 说　　明 |
|---|---|
| Count | 返回栈中的元素个数 |
| Push | 在栈顶添加一个元素 |
| Pop | 从栈顶删除一个元素，并返回该元素。如果栈是空的，就抛出 InvalidOperationException 异常 |
| Peek | 返回栈顶的元素，但不删除它 |
| Contains | 确定某个元素是否在栈中，如果是，就返回 true |

在下面的例子中，使用 Push()方法把 3 个元素添加到栈中。在 foreach 方法中，使用 IEnumerable 接口迭代所有的元素。栈的枚举器不会删除元素，它只会逐个返回元素(代码文件 StackSample/Program.cs)。

```
var alphabet = new Stack<char>();
alphabet.Push('A');
alphabet.Push('B');
alphabet.Push('C');
foreach (char item in alphabet)
{
  Console.Write(item);
}
Console.WriteLine();
```

因为元素的读取顺序是从最后一个添加到栈中的元素开始到第一个元素，所以得到的结果如下：

```
CBA
```

用枚举器读取元素不会改变元素的状态。使用 Pop()方法会从栈中读取每个元素，然后删除它们。这样，就可以使用 while 循环迭代集合，检查 Count 属性，确定栈中是否还有元素：

```
var alphabet = new Stack<char>();
alphabet.Push('A');
```

```
alphabet.Push('B');
alphabet.Push('C');
Console.Write("First iteration: ");
foreach (char item in alphabet)
{
  Console.Write(item);
}
Console.WriteLine();
Console.Write("Second iteration: ");
while (alphabet.Count > 0)
{
  Console.Write(alphabet.Pop());
}
Console.WriteLine();
```

结果是两个 CBA，每次迭代对应一个 CBA。在第二次迭代后，栈变空，因为第二次迭代使用了 Pop()方法：

```
First iteration: CBA
Second iteration: CBA
```

# 10.6　链表

LinkedList<T>是一个双向链表，其元素指向它前面和后面的元素，如图 10-3 所示。这样一来，通过移动到下一个元素可以正向遍历整个链表，通过移动到前一个元素可以反向遍历整个链表。

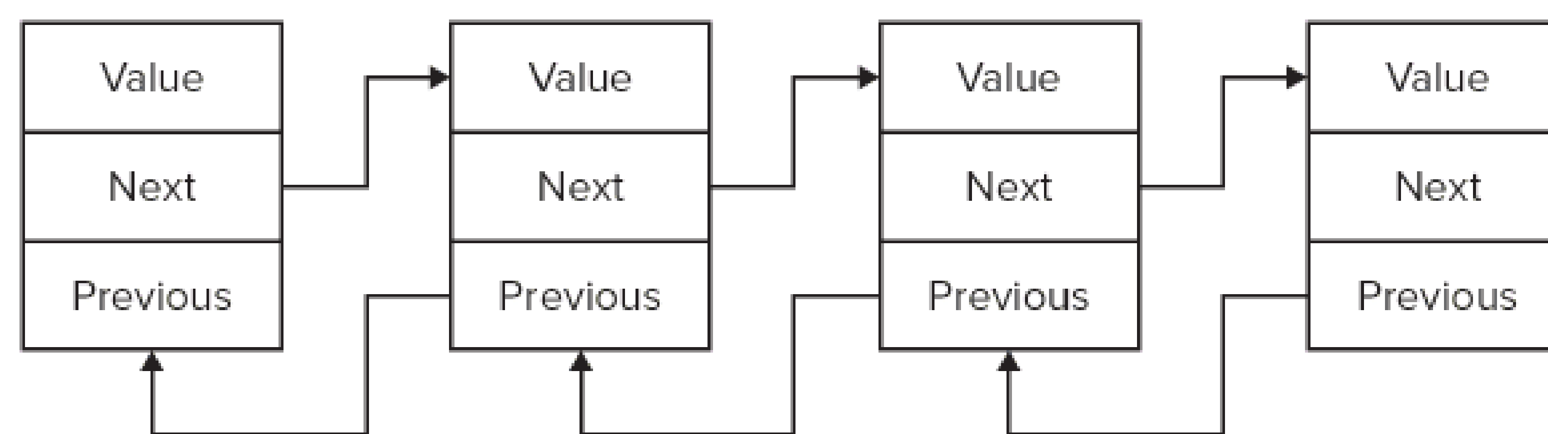



图 10-3

链表的优点是，如果将元素插入列表的中间位置，使用链表就会非常快。在插入一个元素时，只需要修改上一个元素的 Next 引用和下一个元素的 Previous 引用，使它们引用所插入的元素。在 List<T>类中，插入一个元素时，需要移动该元素后面的所有元素。

当然，链表也有缺点。链表的元素只能一个接一个地访问，这需要较长的时间来查找位于链表中间或尾部的元素。

链表不能在列表中仅存储元素。存储元素时，链表还必须存储每个元素的下一个元素和上一个元素的信息。这就是 LinkedList<T>包含 LinkedListNode<T>类型的元素的原因。使用 LinkedListNode<T>类，可以获得列表中的下一个元素和上一个元素。LinkedListNode<T>定义了属性 List、Next、Previous 和 Value。List 属性返回与节点相关的 LinkedList<T>对象，Next 和 Previous 属性用于遍历链表，访问当前节点之后和之前的节点。Value 返回与节点相关的元素，其类型是 T。

LinkedList<T>类定义的成员可以访问链表中的第一个和最后一个元素(First 和 Last)、在指定位置插入元素(AddAfter()、AddBefore()、AddFirst()和 AddLast()方法)，删除指定位置的元素(Remove()、RemoveFirst()和 RemoveLast()方法)、从链表的开头(Find()方法)或结尾(FindLast()方法)开始搜索元素。

示例应用程序使用了一个链表和一个列表。链表包含文档，这与上一个队列例子相同，但文档有一个额外的优先级。在链表中，文档按照优先级来排序。如果多个文档的优先级相同，这些元素就按照文档的插入时间来排序。

图 10-4 描述了示例应用程序中的集合。LinkedList<Document>是一个包含所有 Document 对象的链表，该图显示了文档的标题和优先级。标题指出了文档添加到链表中的时间。第一个添加的文档的标题是“One”。第二个添加的文档的标题是“Two”，依此类推。可以看出，文档 One 和 Four 有相同的优先级 8，因为 One 在 Four 之前添加，所以 One 放在链表的前面。

在链表中添加新文档时，它们应放在优先级相同的最后一个文档后面。集合 LinkedList<Document>包含

LinkedListNode<Document>类型的元素。LinkedListNode<T>类添加 Next 和 Previous 属性，使搜索过程能从一个节点移动到下一个节点上。要引用这类元素，应把 List<T>定义为 List<LinkedListNode<Document>>。为了快速访问每个优先级的最后一个文档，集合 List<LinkedList Node>应最多包含 10 个元素，每个元素分别引用每个优先级的最后一个文档。在后面的讨论中，对每个优先级的最后一个文档的引用称为优先级节点。

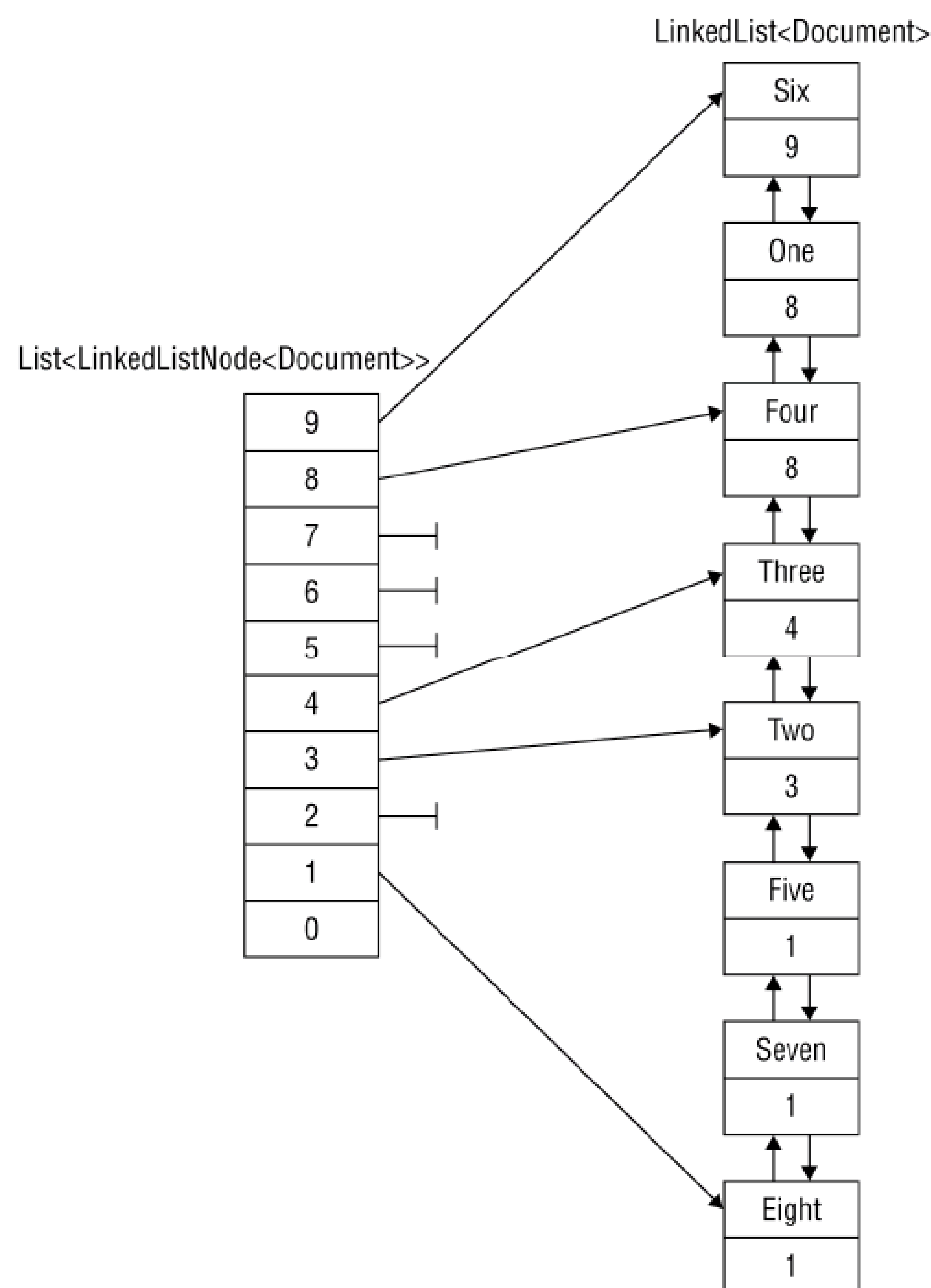



图 10-4

在上面的例子中，Document 类扩展为包含优先级。优先级用类的构造函数设置(代码文件 LinkedListSample/Document.cs)：

```
public class Document
{
  public string Title { get; }
  public string Content { get; }
  public byte Priority { get; }
  public Document(string title, string content, byte priority)
  {
    Title = title;
    Content = content;
    Priority = priority;
  }
}
```

解决方案的核心是 PriorityDocumentManager 类。这个类很容易使用。在这个类的公共接口中，可以把新的 Document 元素添加到链表中，可以检索第一个文档，为了便于测试，它还提供了一个方法，在元素链接到链表中时，该方法可以显示集合中的所有元素。

PriorityDocumentManager 类包含两个集合。LinkedList<Document>类型的集合包含所有的文档。List<LinkedListNode<Document>>类型的集合包含最多 10 个元素的引用，它们是添加指定优先级的新文档的入口点。这两个集合变量都用 PriorityDocumentManager 类的构造函数来初始化。列表集合也用 null 初始化(代码文件 LinkedListSample/PriorityDocumentManager.cs)：

```
public class PriorityDocumentManager
{
```

```
private readonly LinkedList<Document> _documentList;
// priorities 0.9
private readonly List<LinkedListNode<Document>> _priorityNodes;
public PriorityDocumentManager()
{
  _documentList = new LinkedList<Document>();
  _priorityNodes = new List<LinkedListNode<Document>>(10);
  for (int i = 0; i < 10; i++)
  {
    _priorityNodes.Add(new LinkedListNode<Document>(null));
  }
}
```

在类的公共接口中，有一个 AddDocument()方法。AddDocument()方法只调用私有方法 AddDocumentToPriorityNode()。把实现代码放在另一个方法中的原因是，AddDocumentToPriorityNode()方法可以递归调用，如后面所示。

```
public void AddDocument(Document d)
{
  if (d == null) throw new ArgumentNullException(nameof(d));
  AddDocumentToPriorityNode(d, d.Priority);
}
```

在 AddDocumentToPriorityNode()方法的实现代码中，第一个操作是检查优先级是否在允许的优先级范围内。这里允许的范围是 0~9。如果传送了错误的值，就会抛出一个 ArgumentException 类型的异常。

接着检查是否已经有一个优先级节点与所传送的优先级相同。如果在列表集合中没有这样的优先级节点，就递归调用 AddDocumentToPriorityNode()方法，递减优先级值，检查是否有低一级的优先级节点。

如果优先级节点的优先级值与所传送的优先级值不同，也没有比该优先级值更低的优先级节点，就可以调用 AddLast()方法，将文档安全地添加到链表的末尾。另外，链表节点由负责指定文档优先级的优先级节点引用。

如果存在这样的优先级节点，就可以在链表中找到插入文档的位置。这里必须区分是存在指定优先级值的优先级节点，还是存在以较低的优先级值引用文档的优先级节点。对于第一种情况，可以把新文档插入由优先级节点引用的位置后面。因为优先级节点总是引用指定优先级值的最后一个文档，所以必须设置优先级节点的引用。如果引用文档的优先级节点有较低的优先级值，情况就会比较复杂。这里新文档必须插入优先级值与优先级节点相同的所有文档的前面。为了找到优先级值相同的第一个文档，要通过一个 while 循环，使用 Previous 属性遍历所有的链表节点，直到找到一个优先级值不同的链表节点为止。这样，就找到了必须插入文档的位置，并可以设置优先级节点。

```
private void AddDocumentToPriorityNode(Document doc, int priority)
{
  if (priority > 9 || priority < 0)
    throw new ArgumentException("Priority must be between 0 and 9");
  if (_priorityNodes[priority].Value == null)
  {
    --priority;
    if (priority <= 0)
    {
      // check for the next lower priority
      AddDocumentToPriorityNode(doc, priority);
    }
    else // now no priority node exists with the same priority or lower
      // add the new document to the end
    {
      _documentList.AddLast(doc);
      _priorityNodes[doc.Priority] = _documentList.Last;
    }
    return;
  }
  else // a priority node exists
  {
    LinkedListNode<Document> prioNode = _priorityNodes[priority];
    if (priority == doc.Priority)
    // priority node with the same priority exists
    {
      _documentList.AddAfter(prioNode, doc);
      // set the priority node to the last document with the same priority
      _priorityNodes[doc.Priority] = prioNode.Next;
    }
```

```
      else // only priority node with a lower priority exists
      {
        // get the first node of the lower priority
        LinkedListNode<Document> firstPrioNode = prioNode;
        while (firstPrioNode.Previous != null &&
          firstPrioNode.Previous.Value.Priority == prioNode.Value.Priority)
        {
          firstPrioNode = prioNode.Previous;
          prioNode = firstPrioNode;
        }
        _documentList.AddBefore(firstPrioNode, doc);
        // set the priority node to the new value
        _priorityNodes[doc.Priority] = firstPrioNode.Previous;
      }
    }
  }
```

现在还剩下几个简单的方法没有讨论。DisplayAllNodes()方法只是在一个 foreach 循环中，把每个文档的优先级和标题显示在控制台上。

GetDocument()方法从链表中返回第一个文档(优先级最高的文档)，并从链表中删除它：

```
public void DisplayAllNodes()
{
  foreach (Document doc in documentList)
  {
    Console.WriteLine($"priority: {doc.Priority}, title {doc.Title}");
  }
}

// returns the document with the highest priority
// (that's first in the linked list)
public Document GetDocument()
{
  Document doc = _documentList.First.Value;
  _documentList.RemoveFirst();
  return doc;
}
```

在 Main()方法中，PriorityDocumentManager 类用于说明其功能。在链表中添加 8 个优先级不同的新文档，再显示整个链表(代码文件 LinkedListSample/Program.cs)：

```
public static void Main()
{
  var pdm = new PriorityDocumentManager();
  pdm.AddDocument(new Document("one", "Sample", 8));
  pdm.AddDocument(new Document("two", "Sample", 3));
  pdm.AddDocument(new Document("three", "Sample", 4));
  pdm.AddDocument(new Document("four", "Sample", 8));
  pdm.AddDocument(new Document("five", "Sample", 1));
  pdm.AddDocument(new Document("six", "Sample", 9));
  pdm.AddDocument(new Document("seven", "Sample", 1));
  pdm.AddDocument(new Document("eight", "Sample", 1));
  pdm.DisplayAllNodes();
}
```

在处理好的结果中，文档先按优先级排序，再按添加文档的时间排序：

```
priority: 9, title six
priority: 8, title one
priority: 8, title four
priority: 4, title three
priority: 3, title two
priority: 1, title five
priority: 1, title seven
priority: 1, title eight
```

## 10.7　有序列表

如果需要基于键对所需集合排序，就可以使用 SortedList<TKey, TValue>类。这个类按照键给元素排序。这个集合中的值和键都可以使用任意类型。

下面的例子创建了一个有序列表，其中键和值都是 string 类型。默认的构造函数创建了一个空列表，再用

Add()方法添加两本书。使用重载的构造函数，可以定义列表的容量，传递实现了 IComparer<TKey>接口的对象，该接口用于给列表中的元素排序。

Add()方法的第一个参数是键(书名)，第二个参数是值(ISBN 号)。除了使用 Add()方法之外，还可以使用索引器将元素添加到列表中。索引器需要把键作为索引参数。如果键已存在，Add()方法就抛出一个 ArgumentException 类型的异常。如果索引器使用相同的键，就用新值替代旧值(代码文件 SortedListSample/Program.cs)。

```
var books = new SortedList<string, string>();
books.Add("Professional WPF Programming", "978-0-470-04180-2");
books.Add("Professional ASP.NET MVC 5", "978-1-118-79475-3");

books["Beginning C# 6 Programming"] = "978-1-119-09668-9";
books["Professional C# 6 and .NET Core 1.0"] = "978-1-119-09660-3";
```

**注意：**
SortedList<TKey, TValue>类只允许每个键有一个对应的值，如果需要每个键对应多个值，就可以使用 Lookup<TKey, TElement>类。

可以使用 foreach 语句遍历该列表。枚举器返回的元素是 KeyValuePair<TKey, TValue>类型，其中包含了键和值。键可以用 Key 属性访问，值可以用 Value 属性访问。

```
foreach (KeyValuePair<string, string> book in books)
{
  Console.WriteLine($"{book.Key}, {book.Value}");
}
```

迭代语句会按键的顺序显示书名和 ISBN 号：

```
Beginning C# 6 Programming, 978-1-119-09668-9
Professional ASP.NET MVC 5, 978-1-118-79475-3
Professional C# 6 and .NET Core 1.0, 978-1-119-09660-3
Professional WPF Programming, 978-0-470-04180-2
```

也可以使用 Values 和 Keys 属性访问值和键。因为 Values 属性返回 IList<TValue>，Keys 属性返回 IList<TKey>，所以可以通过 foreach 语句使用这些属性：

```
foreach (string isbn in books.Values)
{
  Console.WriteLine(isbn);
}
foreach (string title in books.Keys)
{
  Console.WriteLine(title);
}
```

第一个循环显示值，第二个循环显示键：

```
978-1-119-09668-9
978-1-118-79475-3
978-1-119-09660-3
978-0-470-04180-2
Beginning C# 6 Programming
Professional ASP.NET MVC 5
Professional C# 6 and .NET Core 1.0
Professional WPF Programming
```

如果尝试使用索引器访问一个元素，但所传递的键不存在，就会抛出一个 KeyNotFoundException 类型的异常。为了避免这个异常，可以使用 ContainsKey()方法，如果所传递的键存在于集合中，这个方法就返回 true，也可以调用 TryGetValue()方法，如果指定键对应的值不存在，该方法就会尝试获得指定键的值，而不会抛出异常。

```
string title = "Professional C# 8";
if (!books.TryGetValue(title, out string isbn))
{
  Console.WriteLine($"{title} not found");
}
```

# 10.8　字典

字典表示一种非常复杂的数据结构，这种数据结构允许按照某个键来访问元素。字典也称为映射或散列表。字典的主要特性是能根据键快速查找值。也可以自由地添加和删除元素，这有点像 List<T>类，但没有在内存中移动后续元素的性能开销。

图 10-5 是字典的一个简化表示。其中 employee-id(如 B4711)是添加到字典中的键。键会转换为一个散列。利用散列创建一个数字，它将索引和值关联起来。然后索引包含一个到值的链接。该图做了简化处理，因为一个索引项可以关联多个值，索引可以存储为一个树型结构。

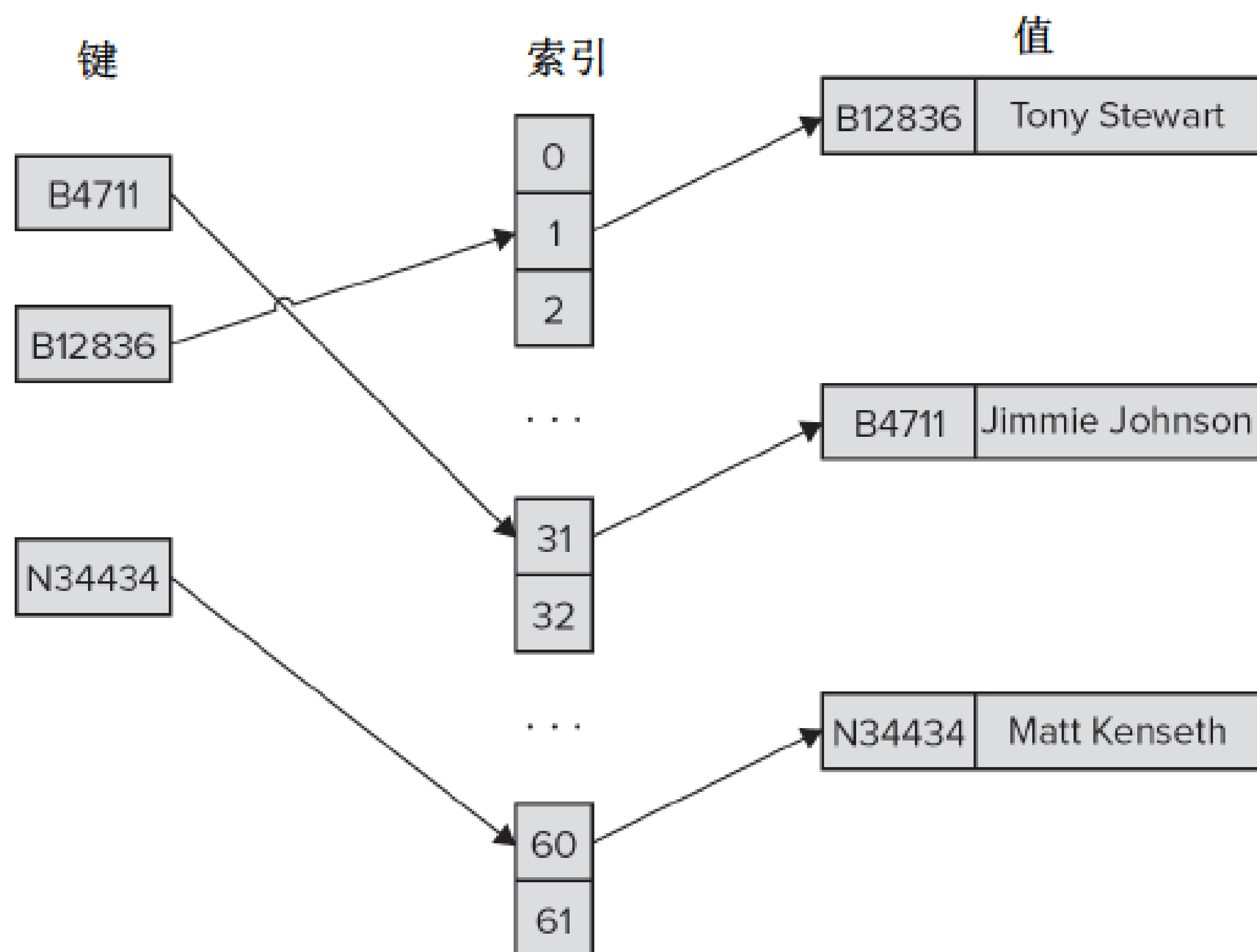



图 10-5

.NET Framework 提供了几个字典类。可以使用的最主要的类是 Dictionary<TKey, TValue>。

## 10.8.1　字典初始化器

C# 提供了一个语法，在声明时初始化字典。带有 int 键和 string 值的字典可以初始化如下：

```
var dict = new Dictionary<int, string>()
{
  [3] = "three",
  [7] = "seven"
};
```

这里把两个元素添加到字典中。第一个元素的键是 3，字符串值是 three；第二个元素的键是 7，字符串值是 seven。这个初始化语法易于阅读，使用的语法与访问字典中的元素相同。

## 10.8.2　键的类型

用作字典中键的类型必须重写 Object 类的 GetHashCode()方法。只要字典类需要确定元素的位置，它就要调用 GetHashCode()方法。GetHashCode()方法返回的 int 由字典用于计算在对应位置放置元素的索引。这里不介绍这个算法。我们只需要知道，它涉及素数，所以字典的容量是一个素数。

GetHashCode()方法的实现代码必须满足如下要求：

- 相同的对象应总是返回相同的值。
- 不同的对象可以返回相同的值。
- 它不能抛出异常。
- 它应至少使用一个实例字段。
- 散列代码最好在对象的生存期中不发生变化。

除了 GetHashCode()方法的实现代码必须满足的要求之外，最好还满足如下要求：

- 它应执行得比较快，计算的开销不大。
- 散列代码值应平均分布在 int 可以存储的整个数字范围上。

**注意：**
字典的性能取决于 GetHashCode()方法的实现代码。

为什么要使散列代码值平均分布在整数的取值范围内？如果两个键返回的散列代码值会得到相同的索引，字典类就必须寻找最近的可用空闲位置来存储第二个数据项，这需要进行一定的搜索，以便以后检索这一项。显然这会降低性能，如果在排序时许多键都有相同的索引，这类冲突就更可能出现。根据 Microsoft 的算法的工作方式，当计算出来的散列代码值平均分布在 int.MinValue 和 int.MaxValue 之间时，这种风险会降低到最小。

除了实现 GetHashCode()方法之外，键类型还必须实现 IEquatable<T>.Equals()方法，或重写 Object 类的 Equals()方法。因为不同的键对象可能返回相同的散列代码，所以字典使用 Equals()方法来比较键。字典检查两个键 A 和 B 是否相等，并调用 A.Equals(B)方法。这表示必须确保下述条件总是成立：

如果 A.Equals(B)方法返回 true，则 A.GetHashCode()和 B.GetHashCode()方法必须总是返回相同的散列代码。

这似乎有点奇怪，但它非常重要。如果设计出某种重写这些方法的方式，使上面的条件并不总是成立，那么把这个类的实例用作键的字典就不能正常工作，而是会发生有趣的事情。例如，把一个对象放在字典中后，就再也检索不到它，或者试图检索某项，却返回了错误的项。

**注意：**
如果为 Equals()方法提供了重写版本，但没有提供 GetHashCode()方法的重写版本，C#编译器就会显示一个编译警告。

对于 System.Object，这个条件为 true，因为 Equals()方法只是比较引用，GetHashCode()方法实际上返回一个仅基于对象地址的散列代码。这说明，如果散列表基于一个键，而该键没有重写这些方法，这个散列表就能正常工作。但是，这么做的问题是，只有对象完全相同，键才被认为是相等的。也就是说，把一个对象放在字典中时，必须将它与该键的引用关联起来。也不能在以后用相同的值实例化另一个键对象。如果没有重写 Equals()方法和 GetHashCode()方法，在字典中使用类型时就不太方便。

另外，System.String 实现了 IEquatable 接口，并重载了 GetHashCode()方法。Equals()方法提供了值的比较，GetHashCode()方法根据字符串的值返回一个散列代码。因此，在字典中把字符串用作键非常方便。

数字类型(如 Int32)也实现 IEquatable 接口，并重载 GetHashCode()方法。但是这些类型返回的散列代码只映射到值上。如果希望用作键的数字本身没有分布在可能的整数值范围内，把整数用作键就不能满足键值的平均分布规则，于是不能获得最佳的性能。Int32 并不适合在字典中使用。

如果需要使用的键类型没有实现 IEquatable 接口，也没有根据存储在字典中的键值重载 GetHashCode()方法，就可以创建一个实现 IEqualityComparer<T>接口的比较器。IEqualityComparer<T>接口定义了 GetHashCode()和 Equals()方法，并将传递的对象作为参数，因此可以提供与对象类型不同的实现方式。Dictionary<TKey, TValue>构造函数的一个重载版本允许传递一个实现了 IEqualityComparer <T>接口的对象。如果把这个对象赋予字典，该类就用于生成散列代码并比较键。

### 10.8.3　字典示例

本节的字典示例程序建立了一个员工字典。该字典用 EmployeeId 对象来索引，存储在字典中的每个数据项都是一个 Employee 对象，该对象存储员工的详细数据。

实现 EmployeeId 结构是为了定义在字典中使用的键，该结构的成员是表示员工的一个前缀字符和一个数字。这两个变量都是只读的，只能在构造函数中初始化。字典中的键不应改变，这是必须保证的。在构造函数中填充字

段。重载ToString()方法是为了获得员工ID的字符串表示。与键类型的要求一样，EmployeeId结构也要实现IEquatable接口，并重载GetHashCode()方法(代码文件DictionarySample/EmployeeId.cs)。

```
public class EmployeeIdException : Exception
{
  public EmployeeIdException(string message) : base(message) { }
}

public struct EmployeeId : IEquatable<EmployeeId>
{
  private readonly char _prefix;
  private readonly int _number;
  public EmployeeId(string id)
  {
    if (id == null) throw new ArgumentNullException(nameof(id));
    _prefix = (id.ToUpper())[0];
    int numLength = id.Length - 1;
    try
    {
      _number = int.Parse(id.Substring(1, numLength > 6 ? 6 : numLength));
    }
    catch (FormatException)
    {
      throw new EmployeeIdException("Invalid EmployeeId format");
    }
  }

  public override string ToString() => _prefix.ToString() +
    $"{number,6:000000}";

  public override int GetHashCode() => (number ^ number << 16) * 0x15051505;

  public bool Equals(EmployeeId other) =>
    (prefix == other?.prefix && number == other?.number);

  public override bool Equals(object obj) => Equals((EmployeeId)obj);

  public static bool operator ==(EmployeeId left, EmployeeId right) =>
    left.Equals(right);

  public static bool operator !=(EmployeeId left, EmployeeId right) =>
    !(left == right);
}
```

由IEquatable<T>接口定义的Equals()方法比较两个EmployeeId对象的值，如果这两个值相同，它就返回true。除了实现IEquatable<T>接口中的Equals()方法之外，还可以重写Object类中的Equals()方法。

```
public bool Equals(EmployeeId other) =>
  prefix == other.prefix && number == other.number;
```

由于数字是可变的，因此员工可以取1～190 000的一个值。这并没有填满整数取值范围。GetHashCode()方法使用的算法将数字向左移动16位，再与原来的数字进行异或操作，最后将结果乘以十六进制数15051505。散列代码在整数取值区域上的分布相当均匀：

```
public override int GetHashCode() => (number ^ number << 16) * 0x1505_1505;
```

**注意：**

*在Internet上，有许多更复杂的算法，它们能使散列代码在整数取值范围上更好地分布。也可以使用字符串的GetHashCode()方法来返回一个散列。*

Employee类是一个简单的实体类，该实体类包含员工的姓名、薪水和ID。构造函数初始化所有值，ToString()方法返回一个实例的字符串表示。ToString()方法的实现代码使用格式化字符串创建字符串表示，以提高性能(代码文件DictionarySample/Employee.cs)。

```
public class Employee
{
  private string _name;
  private decimal _salary;
  private readonly EmployeeId _id;
```

```
    public Employee(EmployeeId id, string name, decimal salary)
    {
      _id = id;
      _name = name;
      _salary = salary;
    }

    public override string ToString() =>
      $"{id.ToString()}: {name, -20} {salary:C}";
  }
```

在示例应用程序的 Main()方法中，创建一个新的 Dictionary<TKey, TValue>实例，其中键是 EmployeeId 类型，值是 Employee 类型。构造函数指定了 31 个元素的容量。注意容量一般是素数。但如果指定了一个不是素数的值，也不需要担心。Dictionary<TKey, TValue>类会使用传递给构造函数的整数后面紧接着的一个素数来指定容量。创建员工对象和 ID 后，就使用新的字典初始化语法把它们添加到新建的字典中。当然，也可以调用字典的 Add()方法添加对象(代码文件 DictionarySample/Program.cs)：

```
static void Main()
{
  var idJimmie = new EmployeeId("C48");
  var jimmie = new Employee(idJimmie, "Jimmie Johnson", 150926.00m);

  var idJoey = new EmployeeId("F22");
  var joey = new Employee(idJoey, "Joey Logano", 45125.00m);

  var idKyle = new EmployeeId("T18");
  var kyle = new Employee(idKyle, "Kyle Bush", 78728.00m);

  var idCarl = new EmployeeId("T19");
  var carl = new Employee(idCarl, "Carl Edwards", 80473.00m);

  var idMatt = new EmployeeId("T20");
  var matt = new Employee(idMatt, "Matt Kenseth", 113970.00m);

  var employees = new Dictionary<EmployeeId, Employee>(31)
  {
    [idJimmie] = jimmie,
    [idJoey] = joey,
    [idKyle] = kyle,
    [idCarl] = carl,
    [idMatt] = matt
  };

  foreach (var employee in employees.Values)
  {
    Console.WriteLine(employee);
  }
  //...
```

将数据项添加到字典中后，在 while 循环中读取字典中的员工。让用户输入一个员工号，把该号码存储在变量 userInput 中。用户输入 X 即可退出应用程序。如果输入的键在字典中，就使用 Dictionary<TKey, TValue>类的 TryGetValue()方法检查它。如果找到了该键，TryGetValue()方法就返回 true；否则返回 false。如果找到了与键关联的值，该值就存储在 employee 变量中，并把该值写入控制台。

**注意：**

也可以使用 Dictionary<TKey, TValue>类的索引器替代 TryGetValue()方法，来访问存储在字典中的值。但是，如果没有找到键，索引器会抛出一个 KeyNotFoundException 类型的异常。

```
while (true)
{
  Console.Write("Enter employee id (X to exit)> ");
  var userInput =ReadLine();
  userInput = userInput.ToUpper();
  if (userInput == "X") break;
  EmployeeId id;
  try
  {
    id = new EmployeeId(userInput);
    if (!employees.TryGetValue(id, out Employee employee))
```

```
        {
          Console.WriteLine($"Employee with id {id} does not exist");
        }
        else
        {
          Console.WriteLine(employee);
        }
      }
      catch (EmployeeIdException ex)
      {
        Console.WriteLine(ex.Message);
      }
    }
```

运行应用程序，得到如下输出：

```
C000048: Jimmie Johnson        $150,926.00
F000022: Joey Logano           $45,125.00
T000018: Kyle Bush             $78,728.00
T000019: Carl Edwards          $80,473.00
T000020: Matt Kenseth          $113,970.00
Enter employee id (X to exit)> T18
T000018: Kyle Bush             $78,728.00
Enter employee id (X to exit)> C48
C000048: Jimmie Johnson        $150,926.00
Enter employee id (X to exit)> X
Press any key to continue . . .
```

### 10.8.4　Lookup 类

Dictionary<TKey, TValue>类支持每个键关联一个值。Lookup<TKey, TElement>类非常类似于 Dictionary<TKey, TValue>类，但把键映射到一个值集合上。这个类在程序集 System.Core 中实现，用 System.Linq 名称空间定义。

Lookup<TKey, TElement>类不能像一般的字典那样创建，而必须调用 ToLookup()方法，该方法返回一个 Lookup<TKey, TElement>对象。ToLookup()方法是一个扩展方法，它可以用于实现 IEnumerable<T>接口的所有类。在下面的例子中，填充了一个 Racer 对象列表。因为 List<T>类实现了 IEnumerable<T>接口，所以可以在赛车手列表上调用 ToLookup()方法。这个方法需要一个 Func<TSource, TKey>类型的委托，Func<TSource, TKey>类型定义了键的选择器。这里使用 lambda 表达式 r => r.Country，根据国家来选择赛车手。foreach 循环只使用索引器访问来自澳大利亚的赛车手(代码文件 LookupSample/Program.cs)。

```
var racers = new List<Racer>();
racers.Add(new Racer("Jacques", "Villeneuve", "Canada", 11));
racers.Add(new Racer("Alan", "Jones", "Australia", 12));
racers.Add(new Racer("Jackie", "Stewart", "United Kingdom", 27));
racers.Add(new Racer("James", "Hunt", "United Kingdom", 10));
racers.Add(new Racer("Jack", "Brabham", "Australia", 14));

var lookupRacers = racers.ToLookup(r => r.Country);

foreach (Racer r in lookupRacers["Australia"])
{
  Console.WriteLine(r);
}
```

**注意：**
扩展方法详见第 12 章，lambda 表达式参见第 8 章。

结果显示了来自澳大利亚的赛车手：

```
Alan Jones
Jack Brabham
```

### 10.8.5　有序字典

SortedDictionary<TKey, TValue>是一个二叉搜索树，其中的元素根据键排序。该键类型必须实现

IComparable<TKey>接口。如果键的类型不能排序，则还可以创建一个实现了 IComparer <TKey>接口的比较器，将比较器用作有序字典的构造函数的一个参数。

如前所述，SortedDictionary<TKey, TValue>和 SortedList<TKey, TValue>的功能类似。但因为 SortedList <TKey, TValue>实现为一个基于数组的列表，而 SortedDictionary<TKey, TValue>类实现为一个字典，所以它们有不同的特征。

- SortedList<TKey, TValue>使用的内存比 SortedDictionary<TKey, TValue>少。
- SortedDictionary<TKey, TValue>的元素插入和删除操作比较快。
- 在用已排好序的数据填充集合时，若不需要修改容量，SortedList<TKey, TValue>就比较快。

**注意：**
SortedList 使用的内存比 SortedDictionary 少，但 SortedDictionary 在插入和删除未排序的数据时比较快。

## 10.9 集

包含不重复元素的集合称为“集(set)”。.NET Core 包含两个集(HashSet<T>和 SortedSet<T>)，它们都实现 ISet<T>接口。HashSet<T>集包含不重复元素的无序列表，SortedSet<T>集包含不重复元素的有序列表。

ISet<T>接口提供的方法可以创建合集、交集，或者给出一个集是另一个集的超集或子集的信息。

在下面的示例代码中，创建了 3 个字符串类型的新集，并用一级方程式汽车填充它们。HashSet<T>集实现 ICollection<T>接口。但是在该类中，Add()方法是显式实现的，还提供了另一个 Add()方法。Add()方法的区别是返回类型，它返回一个布尔值，说明是否添加了元素。如果该元素已经在集中，就不添加它，并返回 false(代码文件 SetSample/Program.cs)。

```
var companyTeams = new HashSet<string>()
{ "Ferrari", "McLaren", "Mercedes" };

var traditionalTeams = new HashSet<string>() { "Ferrari", "McLaren" };

var privateTeams = new HashSet<string>()
{ "Red Bull", "Toro Rosso", "Force India", "Sauber" };

if (privateTeams.Add("Williams"))
{
  Console.WriteLine("Williams added");
}

if (!companyTeams.Add("McLaren"))
{
  Console.WriteLine("McLaren was already in this set");
}
```

两个 Add()方法的输出写到控制台上：

```
Williams added
McLaren was already in this set
```

IsSubsetOf()和 IsSupersetOf()方法比较集和实现了 IEnumerable<T>接口的集合，并返回一个布尔结果。这里，IsSubsetOf()方法验证 traditionalTeams 集合中的每个元素是否都包含在 companyTeams 集合方法中，IsSupersetOf()方法验证 traditionalTeams 集合是否有 companyTeams 集合没有的额外元素。

```
if (traditionalTeams.IsSubsetOf(companyTeams))
{
  Console.WriteLine("traditionalTeams is subset of companyTeams");
}
if (companyTeams.IsSupersetOf(traditionalTeams))
{
  Console.WriteLine("companyTeams is a superset of traditionalTeams");
}
```

这个验证的结果如下：

```
traditionalTeams is a subset of companyTeams
companyTeams is a superset of traditionalTeams
```

Williams 也是一个传统队，因此这个队添加到 traditionalTeams 集合中：

```
traditionalTeams.Add("Williams");
if (privateTeams.Overlaps(traditionalTeams))
{
  Console.WriteLine("At least one team is the same with traditional " +
    "and private teams");
}
```

因为有一个重叠，所以结果如下：

```
At least one team is the same with traditional and private teams.
```

调用 UnionWith()方法，把引用新 SortedSet<string>的变量 allTeams 填充为 companyTeams、privateTeams 和 traditionalTeams 的合集：

```
var allTeams = new SortedSet<string>(companyTeams);
allTeams.UnionWith(privateTeams);
allTeams.UnionWith(traditionalTeams);
Console.WriteLine();
Console.WriteLine("all teams");
foreach (var team in allTeams)
{
  Console.WriteLine(team);
}
```

这里返回所有队，但每个队都只列出一次，因为集只包含唯一值。因为容器是 SortedSet<string>，所以结果是有序的：

```
Ferrari
Force India
Lotus
McLaren
Mercedes
Red Bull
Sauber
Toro Rosso
Williams
```

ExceptWith()方法从 allTeams 集中删除所有私有队：

```
allTeams.ExceptWith(privateTeams);
WriteLine();
WriteLine("no private team left");
foreach (var team in allTeams)
{
  Console.WriteLine(team);
}
```

集合中的其他元素不包含私有队：

```
Ferrari
McLaren
Mercedes
```

## 10.10 性能

许多集合类都提供了相同的功能，例如，SortedList 类与 SortedDictionary 类的功能几乎完全相同。但是，其性能常常有很大区别。一个集合使用的内存少，另一个集合的元素检索速度快。在 MSDN 文档中，集合的方法常常有性能提示，给出了以大写 O 记号表示的操作时间：

- O(1)
- O(log n)
- O(n)

O(1)表示无论集合中有多少数据项，这个操作需要的时间都不变。例如，ArrayList 类的 Add()方法就具有 O(1)行为。无论列表中有多少个元素，在列表末尾添加一个新元素的时间都相同。Count 属性会给出元素个数，所以很容易找到列表末尾。

O(n)表示对于集合执行一个操作需要的时间在最坏情况时是 N。如果需要重新给集合分配内存，ArrayList 类的 Add()方法就是一个 O(n)操作。改变容量，需要复制列表，复制的时间随元素的增加而线性增加。

O(log n)表示操作需要的时间随集合中元素的增加而增加，但每个元素需要增加的时间不是线性的，而是呈对数曲线。在集合中执行插入操作时，SortedDictionary<TKey,TValue>集合类具有 O(log n)行为，而 SortedList<TKey,TValue>集合类具有 O(n)行为。这里 SortedDictionary <TKey,TValue>集合类要快得多，因为它在树型结构中插入元素的效率比列表高得多。

表 10-4 列出了集合类及其执行不同操作的性能，例如，添加、插入和删除元素。使用这个表可以选择性能最佳的集合类。左列是集合类，Add 列给出了在集合中添加元素所需的时间。List<T>和 HashSet<T>类把 Add 方法定义为在集合中添加元素。其他集合类用不同的方法把元素添加到集合中。例如，Stack<T>类定义了 Push()方法，Queue<T>类定义了 Enqueue()方法。这些信息也列在表中。

如果单元格中有多个大 O 值，表示若集合需要重置大小，该操作就需要一定的时间。例如，在 List<T>类中，添加元素的时间是 O(1)。如果集合的容量不够大，需要重置大小，则重置大小需要的时间长度就是 O(n)。集合越大，重置大小操作的时间就越长。最好避免重置集合的大小，而应把集合的容量设置为一个可以包含所有元素的值。

如果表单元格的内容是 n/a(代表 not applicable)，就表示这个操作不能应用于这种集合类型。

表 10-4

| 集　合 | Add | Insert | Remove | Item | Sort | Find |
|---|---|---|---|---|---|---|
| List<T> | 如果集合必须重置大小，就是 O(1)或 O(n) | O(n) | O(n) | O(1) | O(n log n)，最坏的情况是 O (n ^ 2) | O(n) |
| Stack<T> | Push()，如果栈必须重置大小，就是 O(1)或 O(n) | n/a | Pop, O(1) | n/a | n/a | n/a |
| Queue<T> | Enqueue()，如果队列必须重置大小，就是 O(1)或 O(n) | n/a | Dequeue, O(1) | n/a | n/a | n/a |
| HashSet<T> | 如果集必须重置大小，就是 O(1)或 O(n) | AddO(1)或 O(n) | O(1) | n/a | n/a | n/a |
| SortedSet<T> | 如果集必须重置大小，就是 O(1)或 O(n) | AddO(1)或 O(n) | O(1) | n/a | n/a | n/a |
| LinkedList<T> | AddLastO(1) | AddAfter O(1) | O(1) | n/a | n/a | O(n) |
| Dictionary <TKey, TValue> | O(1)或 O(n) | n/a | O(1) | O(1) | n/a | n/a |
| SortedDictionary <TKey, TValue> | O(log n) | n/a | O(log n) | O(log n) | n/a | n/a |
| SortedList <TKey, TValue> | 无序数据为 O(n)；如果必须重置大小，就是 O(n)；到列表的尾部，就是 O(log n) | n/a | O(n) | 读/写是 O(log n)；如果键在列表中，就是 O(log n)；如果键不在列表中，就是 O(n) | n/a | n/a |

## 10.11　小结

本章介绍了如何处理不同类型的泛型集合。数组的大小是固定的，但可以使用列表作为动态增长的集合。队列以先进先出的方式访问元素，栈以后进先出的方式访问元素。链表可以快速地插入和删除元素，但搜索操作比较慢。通过键和值可以使用字典，它的搜索和插入操作比较快。集(set)用于唯一项，可以是无序的(HashSet<T>)，也可以是有序的(SortedSet<T>)。

第 11 章将介绍一些特殊的集合类。

# 第11章

# 特殊的集合

**本章要点**

- 使用位数组和位矢量
- 使用可观察的集合
- 使用不可变的集合
- 使用并发的集合

**本章源代码下载地址(wrox.com)：**

打开 www.wrox.com 的 Download Code 选项卡可下载本章源代码。源代码也可以在 SpecialCollections 目录的 https://github.com/ProfessionalCSharp/ProfessionalCSharp7 中找到。本章代码分为以下几个主要的示例文件：

- 位数组示例
- 位矢量示例
- 可观察的集合示例
- 不可变的集合示例
- 管道示例

## 11.1 概述

第 10 章介绍了列表、队列、堆栈、字典和链表。本章继续介绍特殊的集合，例如，处理位的集合、改变时可以观察的集合、不能改变的集合，以及可以在多个线程中同时访问的集合。

## 11.2 处理位

如果需要处理的数字有许多位，C# 7 为此提供了二进制字面量和数字分隔符，参见第 2 章和第 6 章。处理二进制数据时，还可以使用 BitArray 类和 BitVector32 结构。BitArray 类位于名称空间 System.Collections 中，BitVector32 结构位于名称空间 System.Collections.Specialized 中。这两种类型最重要的区别是，BitArray 类可以重新设置大小，如果事先不知道需要的位数，就可以使用 BitArray 类，它可以包含非常多的位。BitVector32 结构是基于栈的，因此比较快。BitVector32 结构仅包含 32 位，它们存储在一个整数中。

### 11.2.1　BitArray 类

BitArray 类是一个引用类型，它包含一个 int 数组，其中每 32 位使用一个新整数。这个类的成员如表 11-1 所示。

表 11-1

| BitArray 类的成员 | 说　明 |
|---|---|
| Count<br>Length | Count 和 Length 属性的 get 访问器返回数组中的位数。使用 Length 属性还可以定义新的数组大小，重新设置集合的大小 |
| Item<br>Get<br>Set | 可以使用索引器读写数组中的位。索引器是布尔类型。除了使用索引器之外，还可以使用 Get()和 Set()方法访问数组中的位 |
| SetAll | 根据传送给该方法的参数，SetAll()方法设置所有位的值 |
| Not | Not()方法对数组中所有位的值取反 |
| And<br>Or<br>Xor | 使用 And()、Or()和 Xor()方法，可以合并两个 BitArray 对象。And()方法执行二元 AND，只有两个输入数组的位都设置为 1，结果位才是 1。Or()方法执行二元 OR，只要有一个输入数组的位设置为 1，结果位就是 1。Xor()方法是异或操作，只有一个输入数组的位设置为 1，结果位才是 1 |

**注意：**

第 6 章中介绍了按位运算符，它可以用于数字类型(如 byte、short、int 和 long)。BitArray 类具有类似的功能，但是可以用于不同数量的位，而不是用于 C#类型。

BitArraySample 使用如下名称空间：

System

System.Collections

System.Text

扩展方法 GetBitsFormat()遍历 BitArray，根据位的设置情况，在控制台上显示 1 或 0。为了获得更好的可读性，每 4 位添加了一个分隔符(代码文件 BitArraySample/Program.cs)：

```
public static class BitArrayExtensions
{
  public static string GetBitsFormat(this BitArray bits)
  {
    var sb = new StringBuilder();
    for (int i = bits.Length - 1; i >= 0; i--)
    {
      sb.Append(bits[i] ? 1 : 0);
      if (i != 0 && i % 4 == 0)
      {
        sb.Append("_");
      }
    }
    return sb.ToString();
  }
}
```

说明 BitArray 类的示例创建了一个包含 9 位的数组，其索引是 0~8。SetAll()方法把这 9 位都设置为 true。接着 Set()方法把对应于 1 的位设置为 false。除了 Set()方法之外，还可以使用索引器，例如，下面的第 5 个和第 7 个索引(代码文件 BitArraySample/Program.cs)：

```
var bits1 = new BitArray(9);
bits1.SetAll(true);
bits1.Set(1, false);
```

```
bits1[5] = false;
bits1[7] = false;
Console.Write("initialized: ");
Console.WriteLine(bits1.GetBitsFormat());
```

这是初始化位的显示结果：

```
initialized: 1_0101_1101
```

Not()方法会对 BitArray 类的位取反：

```
Console.Write("not ");
Console.Write(bits1.GetBitsFormat());
bits1.Not();
Console.Write(" = ");
Console.WriteLine(bits1.GetBitsFormat());
```

Not()方法的结果是对所有的位取反。如果某位是 true，则执行 Not()方法的结果就是 false，反之亦然。

```
not 1_0101_1101 = 0_1010_0010
```

这里创建了一个新的 BitArray 类。在构造函数中，因为使用变量 bits1 初始化数组，所以新数组与旧数组有相同的值。接着把第 0、1 和 4 位的值设置为不同的值。在使用 Or()方法之前，显示位数组 bits1 和 bits2。Or()方法将改变 bits1 的值：

```
var bits2 = new BitArray(bits1);
bits2[0] = true;
bits2[1] = false;
bits2[4] = true;
Console.Write($"{bits1.GetBitsFormat()} OR {bits2.GetBitsFormat()}");
Console.Write(" = ");
bits1.Or(bits2);
Console.WriteLine(bits1.GetBitsFormat());
```

使用 Or()方法时，从两个输入数组中提取设置位。结果是，如果某位在第一个或第二个数组中设置为 true，该位在执行 Or()方法后就是 true：

```
0_1010_0010 OR 0_1011_0001 = 0_1011_0011
```

下面使用 And()方法作用于位数组 bits1 和 bits2：

```
Console.Write($"{bits2.GetBitsFormat()} AND {bits1.GetBitsFormat()}");
Console.Write(" = ");
bits2.And(bits1);
Console.WriteLine(bits2.GetBitsFormat());
```

And()方法只把在两个输入数组中都设置为 true 的位设置为 true：

```
0_1011_0001 AND 0_1011_0011 = 0_1011_0001
```

最后使用 Xor()方法进行异或操作：

```
Console.Write($"{bits1.GetBitsFormat()} XOR {bits2.GetBitsFormat()}");
bits1.Xor(bits2);
Console.Write(" = ");
Console.WriteLine(bits1.GetBitsFormat());
```

使用 Xor()方法，只有一个(不能是两个)输入数组的位设置为 1，结果位才是 1。

```
0_1011_0011 XOR 0_1011_0001 = 0_0000_0010
```

### 11.2.2 BitVector32 结构

如果事先知道需要的位数，就可以使用 BitVector32 结构替代 BitArray 类。BitVector32 结构效率较高，因为它是一个值类型，在整数栈上存储位。一个整数可以存储 32 位。如果需要更多的位，就可以使用多个 BitVector32 值或 BitArray 类。BitArray 类可以根据需要增大，但 BitVector32 结构不能。

表 11-2 列出了 BitVector32 结构中与 BitArray 类完全不同的成员。

表 11-2

| BitVector32 结构的成员 | 说　明 |
|---|---|
| Data | Data 属性把 BitVector32 结构中的数据返回为整数 |
| Item | BitVector32 的值可以使用索引器设置。索引器是重载的——可以使用掩码或 BitVector32. Section 类型的片段来获取和设置值 |
| CreateMask | 这是一个静态方法，用于为访问 BitVector32 结构中的特定位创建掩码 |
| CreateSection | 这是一个静态方法，用于创建 32 位中的几个片段 |

BitVectorSample 使用如下名称空间：

System

System.Collections.Specialized

System.Linq

示例代码用默认构造函数创建了一个 BitVector32 结构，其中所有的 32 位都初始化为 false。接着创建掩码，以访问位矢量中的位。对 CreateMask()方法的第一个调用创建了用来访问第一位的一个掩码。调用 CreateMask()方法后，bit1 被设置为 1。再次调用 CreateMask()方法，把第一个掩码作为参数传递给 CreateMask()方法，返回用来访问第二位(它是 2)的一个掩码。接着，将 bit3 设置为 4，以访问位编号 3。bit4 的值是 8，以访问位编号 4。

然后，使用掩码和索引器访问位矢量中的位，并相应地设置字段(代码文件 BitArraySample/Program.cs)：

```
var bits1 = new BitVector32();
int bit1 = BitVector32.CreateMask();
int bit2 = BitVector32.CreateMask(bit1);
int bit3 = BitVector32.CreateMask(bit2);
int bit4 = BitVector32.CreateMask(bit3);
int bit5 = BitVector32.CreateMask(bit4);
bits1[bit1] = true;
bits1[bit2] = false;
bits1[bit3] = true;
bits1[bit4] = true;
bits1[bit5] = true;
Console.WriteLine(bits1);
```

BitVector32 结构有一个重写的 ToString()方法，它不仅显示类名，还显示 1 或 0，来说明位是否设置了，如下所示：

```
BitVector32{00000000000000000000000000011101}
```

除了用 CreateMask()方法创建掩码之外，还可以自己定义掩码，也可以一次设置多位。十六进制值 abcdef 与二进制值 1010 1011 1100 1101 1110 1111 相同。用这个值定义的所有位都设置了：

```
bits1[0xabcdef] = true;
Console.WriteLine(bits1);
```

在输出中可以验证设置的位：

```
BitVector32{00000000101010111100110111101111}
```

把 32 位分别放在不同的片段中非常有用。例如，IPv4 地址定义为一个 4 字节的数，该数存储在一个整数中。可以定义 4 个片段，把这个整数拆分开。在多播 IP 消息中，使用了几个 32 位的值。其中一个 32 位的值放在这些片段中：16 位表示源号，8 位表示查询器的查询内部码，3 位表示查询器的健壮变量，1 位表示抑制标志，还有 4 个保留位。也可以定义自己的位含义，以节省内存。

下面的例子模拟接收到值 0x79abcdef，把这个值传送给 BitVector32 结构的构造函数，从而相应地设置位：

```
int received = 0x79abcdef;
BitVector32 bits2 = new BitVector32(received);
Console.WriteLine(bits2);
```

在控制台上显示了初始化的位：

```
BitVector32{01111001101010111100110111101111}
```

接着创建6个片段。第一个片段需要11位，由十六进制值0xfff定义(设置了11位)。片段B需要8位，片段C需要4位，片段D和E需要3位，片段F需要两位。第一次调用CreateSection()方法只是接收0xfff，为最前面的11位分配内存。第二次调用CreateSection()方法时，将第一个片段作为参数传递，从而使下一个片段从第一个片段的结尾处开始。CreateSection()方法返回一个BitVector32. Section类型的值，该类型包含了该片段的偏移量和掩码。

```
// sections: FF EEE DDD CCCC BBBBBBBB
// AAAAAAAAAAAA
BitVector32.Section sectionA = BitVector32.CreateSection(0xfff);
BitVector32.Section sectionB = BitVector32.CreateSection(0xff, sectionA);
BitVector32.Section sectionC = BitVector32.CreateSection(0xf, sectionB);
BitVector32.Section sectionD = BitVector32.CreateSection(0x7, sectionC);
BitVector32.Section sectionE = BitVector32.CreateSection(0x7, sectionD);
BitVector32.Section sectionF = BitVector32.CreateSection(0x3, sectionE);
```

把一个 BitVector32.Section 类型的值传递给 BitVector32 结构的索引器，会返回一个 int，它映射到位矢量的片段上。这里使用一个帮助方法 IntToBinaryString()，获得该 int 数的字符串表示：

```
Console.WriteLine($"Section A: {bits2[sectionA].ToBinaryString()}");
Console.WriteLine($"Section B: {bits2[sectionB].ToBinaryString()}");
Console.WriteLine($"Section C: {bits2[sectionC].ToBinaryString()}");
Console.WriteLine($"Section D: {bits2[sectionD].ToBinaryString()}");
Console.WriteLine($"Section E: {bits2[sectionE].ToBinaryString()}");
Console.WriteLine($"Section F: {bits2[sectionF].ToBinaryString()}");
```

IntToBinaryString()方法接收整数中的位，并返回一个包含 0 和 1 的字符串表示。在实现代码中，Convert.ToString 方法使用 toBase 参数值 2 创建一个二进制表示。在 AddSeparators 扩展方法中，利用 string.Join 方法在每 4 位之后插入一个分隔符，并使用 LINQ 方法将数组与字符串组合。下一章将详细介绍 LINQ(代码文件 BitVectorSample/BinaryExtensions.cs)：

```
public static class BinaryExtensions
{
  public static string AddSeparators(this string number) =>
    number.Length <= 4 ? number :
      string.Join("_",
        Enumerable.Range(0, number.Length / 4)
          .Select(i => number.Substring(i * 4, 4)).ToArray());

  public static string ToBinaryString(this int number) =>
    Convert.ToString(number, toBase: 2).AddSeparators();
}
```

结果显示了片段 A～F 的位表示，现在可以用传递给位矢量的值来验证：

```
Section A: 1101_1110_1111
Section B: 1011_1100
Section C: 1010
Section D: 1
Section E: 111
Section F: 1
```

# 11.3 可观察的集合

如果需要集合中的元素何时删除或添加的信息，就可以使用 ObservableCollection<T>类。这个类最初是为 WPF 定义的，这样 UI 就可以得知集合的变化，通用 Windows 应用程序使用它的方式相同。这个类的名称空间是 System.Collections. ObjectModel。

ObservableCollection<T>类派生自 Collection<T>基类，该基类可用于创建自定义集合，并在内部使用 List<T>类。重写基类中的虚方法 SetItem()和 RemoveItem()，以触发 CollectionChanged 事件。这个类的用户可以使用 INotifyCollectionChanged 接口注册这个事件。

ObservableCollectionSample 使用如下名称空间：

System

System.Collections.ObjectModel

System.Collections.Specialized

下面的示例说明了 ObservableCollection<string>()方法的用法，其中给 CollectionChanged 事件注册了 Data_CollectionChanged()方法。把两项添加到末尾，再插入一项，并删除一项(代码文件 ObservableCollectionSample/Program.cs)：

```
var data = new ObservableCollection<string>();
data.CollectionChanged += Data_CollectionChanged;
data.Add("One");
data.Add("Two");
data.Insert(1, "Three");
data.Remove("One");
```

Data_CollectionChanged()方法接收 NotifyCollectionChangedEventArgs，其中包含了集合的变化信息。Action 属性给出了是否添加或删除一项的信息。对于删除的项，会设置 OldItems 属性，列出删除的项。对于添加的项，则设置 NewItems 属性，列出新增的项。

```
public static void Data_CollectionChanged(object sender,
  NotifyCollectionChangedEventArgs e)
{
  Console.WriteLine($"action: {e.Action.ToString()}");
  if (e.OldItems != null)
  {
    Console.WriteLine($"starting index for old item(s): {e.OldStartingIndex}");
    Console.WriteLine("old item(s):");
    foreach (var item in e.OldItems)
    {
      Console.WriteLine(item);
    }
  }
  if (e.NewItems != null)
  {
    Console.WriteLine($"starting index for new item(s): {e.NewStartingIndex}");
    Console.WriteLine("new item(s): ");
    foreach (var item in e.NewItems)
    {
      Console.WriteLine(item);
    }
  }
  Console.WriteLine();
}
```

运行应用程序，输出如下所示。先在集合中添加 One 和 Two 项，显示的 Add 动作的索引是 0 和 1。第 3 项 Three 插入在位置 1 上，所以显示的 Add 动作的索引是 1。最后删除 One 项，显示的 Remove 动作的索引是 0：

```
action: Add
starting index for new item(s): 0
new item(s):
One
action: Add
starting index for new item(s): 1
new item(s):
Two
action: Add
starting index for new item(s): 1
new item(s):
Three
action: Remove
starting index for old item(s): 0
old item(s):
One
```

## 11.4　不变的集合

如果对象可以改变其状态，就很难在多个同时运行的任务中使用。这些集合必须同步。如果对象不能改变其状态，就很容易在多个线程中使用。不能改变的对象称为不变的对象。不能改变的集合称为不变的集合。

**注意：**
*使用多个任务和线程，以及用异步方法编程的主题详见第 21 章和第 15 章。*

比较前一章讨论的只读集合与不可变的集合，它们有一个很大的差别：只读集合利用可变集合的接口。使用这个接口，不能改变集合。然而，如果有人仍然引用可变的集合，它就仍然可以改变。对于不可变的集合，没有人可以改变这个集合。

ImmutableCollectionSample 利用下面的名称空间：

System
System.Collections.Generic
System.Collections.Immutable

下面是一个简单的不变字符串数组。可以用静态的 Create()方法创建该数组，如下所示。Create 方法被重载，这个方法的其他变体允许传送任意数量的元素。注意，这里使用两种不同的类型：非泛型类 ImmutableArray 的 Create 静态方法和 Create()方法返回的泛型 ImmutableArray 结构。在下面的代码中(代码文件 ImmutableCollectionSample/Program.cs)，创建了一个空数组：

```
ImmutableArray<string> a1 = ImmutableArray.Create<string>();
```

空数组没有什么用。ImmutableArray<T>类型提供了添加元素的 Add()方法。但是，与其他集合类相反，Add()方法不会改变不变集合本身，而是返回一个新的不变集合。因此在调用 Add()方法之后，a1 仍是一个空集合，a2 是包含一个元素的不变集合。Add()方法返回新的不变集合：

```
ImmutableArray<string> a2 = a1.Add("Williams");
```

之后，就可以以流畅的方式使用这个 API，一个接一个地调用 Add()方法。变量 a3 现在引用一个不变集合，它包含 4 个元素：

```
ImmutableArray<string> a3 =
  a2.Add("Ferrari").Add("Mercedes").Add("Red Bull Racing");
```

在使用不变数组的每个阶段，都没有复制完整的集合。相反，不变类型使用了共享状态，仅在需要时复制集合。

但是，先填充集合，再将它变成不变的数组会更高效。需要进行一些处理时，可以再次使用可变的集合。此时可以使用不变集合提供的构建器类。

为了说明其操作，先创建一个 Account 类，将此类放在集合中。这种类型本身是不可变的，不能使用只读自动属性来改变(代码文件 ImmutableCollectionSample/Account.cs)：

```
public class Account
{
  public Account(string name, decimal amount)
  {
    Name = name;
    Amount = amount;
  }

  public string Name { get; }
  public decimal Amount { get; }
}
```

接着创建 List<Account>集合，用示例账户填充(代码文件 ImmutableCollectionSample/Program.cs)：

```
var accounts = new List<Account>()
{
  new Account("Scrooge McDuck", 667377678765m),
  new Account("Donald Duck", -200m),
  new Account("Ludwig von Drake", 20000m)
};
```

有了账户集合，可以使用 ToImmutableList 扩展方法创建一个不变的集合。只要打开名称空间 System.Collections.Immutable，就可以使用这个扩展方法：

```
ImmutableList<Account> immutableAccounts = accounts.ToImmutableList();
```

变量 immutableAccounts 可以像其他集合那样枚举，它只是不能改变。

```
foreach (var account in immutableAccounts)
{
  Console.WriteLine($"{account.Name} {account.Amount}");
}
```

不使用 foreach 语句迭代不变的列表，可以使用用 ImmutableList<T>定义的 ForEach()方法。这个方法需要一个 Action< T >委托作为参数，因此可以分配 lambda 表达式：

```
immutableAccounts.ForEach(a => Console.WriteLine($"{a.Name} {a.Amount}"));
```

为了处理这些集合，可以使用 Contains、FindAll、FindLast、IndexOf 等方法。因为这些方法类似于第 10 章讨论的其他集合类中的方法，所以这里不讨论它们。

如果需要更改不变集合的内容，集合提供了 Add、AddRange、Remove、RemoveAt、RemoveRange、Replace 以及 Sort 方法。这些方法非常不同于正常的集合类，因为用于调用方法的不可变集合永远不会改变，但是这些方法返回一个新的不可变集合。

## 11.4.1　使用构建器和不变的集合

从现有的集合中创建新的不变集合，很容易使用前述的 Add、Remove 和 Replace 方法完成。然而，如果需要进行多个修改，如在新集合中添加和删除元素，这就不是非常高效。为了通过进行更多的修改来创建新的不变集合，可以创建一个构建器。

下面继续前面的示例代码，对集合中的账户对象进行多个更改。为此，可以调用 ToBuilder 方法创建一个构建器。该方法返回一个可以改变的集合。在示例代码中，移除金额大于 0 的所有账户。原来的不变集合没有改变。用构建器进行的改变完成后，调用 Builder 的 ToImmutable 方法，创建一个新的不可变集合。下面使用这个集合输出所有透支账户：

```
ImmutableList<Account>.Builder builder = immutableAccounts.ToBuilder();
for (int i = 0; i > builder.Count; i++)
{
  Account a = builder[i];
  if (a.Amount < 0)
  {
    builder.Remove(a);
  }
}
ImmutableList<Account> overdrawnAccounts = builder.ToImmutable();
overdrawnAccounts.ForEach(a =< WriteLine($"{a.Name} {a.Amount}"));
```

除了使用 Remove 方法删除元素之外，Builder 类型还提供了方法 Add、AddRange、Insert、RemoveAt、RemoveAll、Reverse 以及 Sort，来改变可变的集合。完成可变的操作后，调用 ToImmutable，再次得到不变的集合。

## 11.4.2　不变集合类型和接口

除了 ImmutableArray 和 ImmutableList 之外，NuGet 包 System.Collections.Immutable 还提供了一些不变的集合类型，如表 11-3 所示。

表 11-3

| 不变的集合类型 | 说　明 |
| --- | --- |
| ImmutableArray<T> | ImmutableArray < T >是一个结构，它在内部使用数组类型，但不允许更改底层类型。这个结构实现了接口 IImmutableList < T > |
| ImmutableList<T> | ImmutableList < T >在内部使用一个二叉树来映射对象，以实现接口 IImmutableList < T > |
| ImmutableQueue<T> | IImmutableQueue < T >实现了接口 IImmutableQueue < T >，允许用 Enqueue、Dequeue 和 Peek 以先进先出的方式访问元素 |
| ImmutableStack<T> | ImmutableStack<T>实现了接口 IImmutableStack<T>，允许用 Push、Pop 和 Peek 以先进后出的方式访问元素 |

(续表)

| 不变的集合类型 | 说　明 |
|---|---|
| ImmutableDictionary<TKey,TValue> | ImmutableDictionary < TKey, TValue >是一个不可变的集合，其无序的键/值对元素实现了接口 IImmutableDictionary < TKey, TValue > |
| ImmutableSortedDictionary<TKey, TValue> | ImmutableSortedDictionary < TKey, TValue >是一个不可变的集合，其有序的键/值对元素实现了接口 IImmutableDictionary < TKey, TValue > |
| ImmutableHashSet<T> | ImmutableHashSet < T >是一个不可变的无序散列集，实现了接口 IImmutableSet < T >。该接口提供了第 10 章讨论的功能 |
| ImmutableSortedSet<T> | ImmutableSortedSet < T >是一个不可变的有序集合，实现了接口 IImmutableSet < T > |

与正常的集合类一样，不变的集合也实现了接口，例如，IImmutableQueue<T>、IImmutableList<T>以及IImmutableStack<T>。这些不变接口的最大区别是所有改变集合的方法都返回一个新的集合。

### 11.4.3 使用 LINQ 和不变的数组

为了使用 LINQ 和不变的数组，类 ImmutableArrayExtensions 定义了 LINQ 方法的优化版本，例如，Where、Aggregate、All、First、Last、Select 和 SelectMany。要使用优化的版本，只需要直接使用 ImmutableArray 类型，打开 System.Linq 名称空间。

使用 ImmutableArrayExtensions 类型定义的 Where 方法如下所示，扩展了 ImmutableArray<T>类型：

```
public static IEnumerable<T> Where<T>(
this ImmutableArray<T> immutableArray, Func<T, bool> predicate);
```

正常的 LINQ 扩展方法扩展了 IEnumerable <T>。因为 ImmutableArray <T>是一个更好的匹配，所以使用优化版本调用 LINQ 方法。

**注意：**
LINQ 参见第 12 章。

## 11.5 并发集合

不变的集合很容易在多个线程中使用，因为它们不能改变。如果希望使用应在多个线程中改变的集合，.NET 在名称空间 System.Collections.Concurrent 中提供了几个线程安全的集合类。线程安全的集合可防止多个线程以相互冲突的方式访问集合。

为了对集合进行线程安全的访问，定义了 IProducerConsumerCollection<T>接口。这个接口中最重要的方法是 TryAdd()和 TryTake()。TryAdd()方法尝试给集合添加一项，但如果集合禁止添加项，这个操作就可能失败。为了给出相关信息，TryAdd()方法返回一个布尔值，以说明操作是成功还是失败。TryTake()方法也以这种方式工作，以通知调用者操作是成功还是失败，并在操作成功时返回集合中的项。下面列出了 System.Collections.Concurrent 名称空间中的类及其功能。

- ConcurrentQueue<T>——这个集合类用一种免锁定的算法实现，使用在内部合并到一个链表中的 32 项数组。访问队列元素的方法有 Enqueue()、TryDequeue()和 TryPeek()。这些方法的命名非常类似于前面 Queue<T>类的方法，只是给可能调用失败的方法加上了前缀 Try。因为这个类实现了 IProducerConsumerCollection<T>接口，所以 TryAdd()和 TryTake()方法仅调用 Enqueue()和 TryDequeue()方法。
- ConcurrentStack<T>——非常类似于 ConcurrentQueue<T>类，只是带有另外的元素访问方法。ConcurrentStack<T>类定义了 Push()、PushRange()、TryPeek()、TryPop()以及 TryPopRange()方法。在内部这个类使用其元素的链表。

- ConcurrentBag<T>——该类没有定义添加或提取项的任何顺序。这个类使用一个把线程映射到内部使用的数组上的概念，因此尝试减少锁定。访问元素的方法有 Add()、TryPeek()和 TryTake()。
- ConcurrentDictionary<TKey, TValue>——这是一个线程安全的键值集合。TryAdd()、TryGetValue()、TryRemove()和 TryUpdate()方法以非阻塞的方式访问成员。因为元素基于键和值，所以 ConcurrentDictionary<TKey, TValue>没有实现 IProducerConsumerCollection<T>。
- BlockingCollection<T>——这个集合在可以添加或提取元素之前，会阻塞线程并一直等待。BlockingCollection<T>集合提供了一个接口，以使用 Add()和 Take()方法来添加和删除元素。这些方法会阻塞线程，一直等到任务可以执行为止。 Add()方法有一个重载版本，其中可以给该重载版本传递一个 CancellationToken 令牌。这个令牌允许取消被阻塞的调用。 如果不希望线程无限期地等待下去，且不希望从外部取消调用，就可以使用 TryAdd()和 TryTake()方法，在这些方法中，也可以指定一个超时值，它表示在调用失败之前应阻塞线程和等待的最长时间。

ConcurrentXXX 集合是线程安全的，如果某个动作不适用于线程的当前状态，它们就返回 false。在继续之前，总是需要确认添加或提取元素是否成功。不能相信集合会完成任务。

BlockingCollection<T>是对实现了 IProducerConsumerCollection<T>接口的任意类的修饰器，它默认使用 ConcurrentQueue<T>类。还可以给构造函数传递任何其他实现了 IProducerConsumer Collection<T>接口的类，例如，ConcurrentBag<T>和 ConcurrentStack<T>。

### 11.5.1　创建管道

将这些并发集合类用于管道是一种很好的应用。一个任务向一个集合类写入一些内容，同时另一个任务从该集合中读取内容。

下面的示例应用程序演示了 BlockingCollection<T>类的用法，使用多个任务形成一个管道。第一个管道如图 11-1 所示。第一阶段的任务读取文件名，并把它们添加到队列中。在这个任务运行的同时，第二阶段的任务已经开始从队列中读取文件名并加载它们的内容。结果被写入另一个队列。第三阶段可以同时启动，读取并处理第二个队列的内容。结果被写入一个字典。

在这个场景中，只有第三阶段完成，并且内容已被最终处理，在字典中得到了完整的结果时，下一个阶段才会开始。图 11-2 显示了接下来的步骤。第四阶段从字典中读取内容，转换数据，然后将其写入队列中。第五阶段在项中添加了颜色信息，然后把它们添加到另一个队列中。最后一个阶段显示了信息。第四阶段到第六阶段也可以并发运行。

Info 类代表由管道维护的项(代码文件 PipelineSample/Info.cs )：

```
public class Info
{
  public Info(string word, int count)
  {
    Word = word;
    Count = count;
  }

  public string Word { get; }
  public int Count { get; }
  public string Color { get; set; }

  public override string ToString() => $"{Count} times: {Word}";
}
```

PipelineSample 使用如下名称空间：

System

System.Collections.Generic

System.Collections.Concurrent

System.IO

System.Linq

System.Threading.Tasks

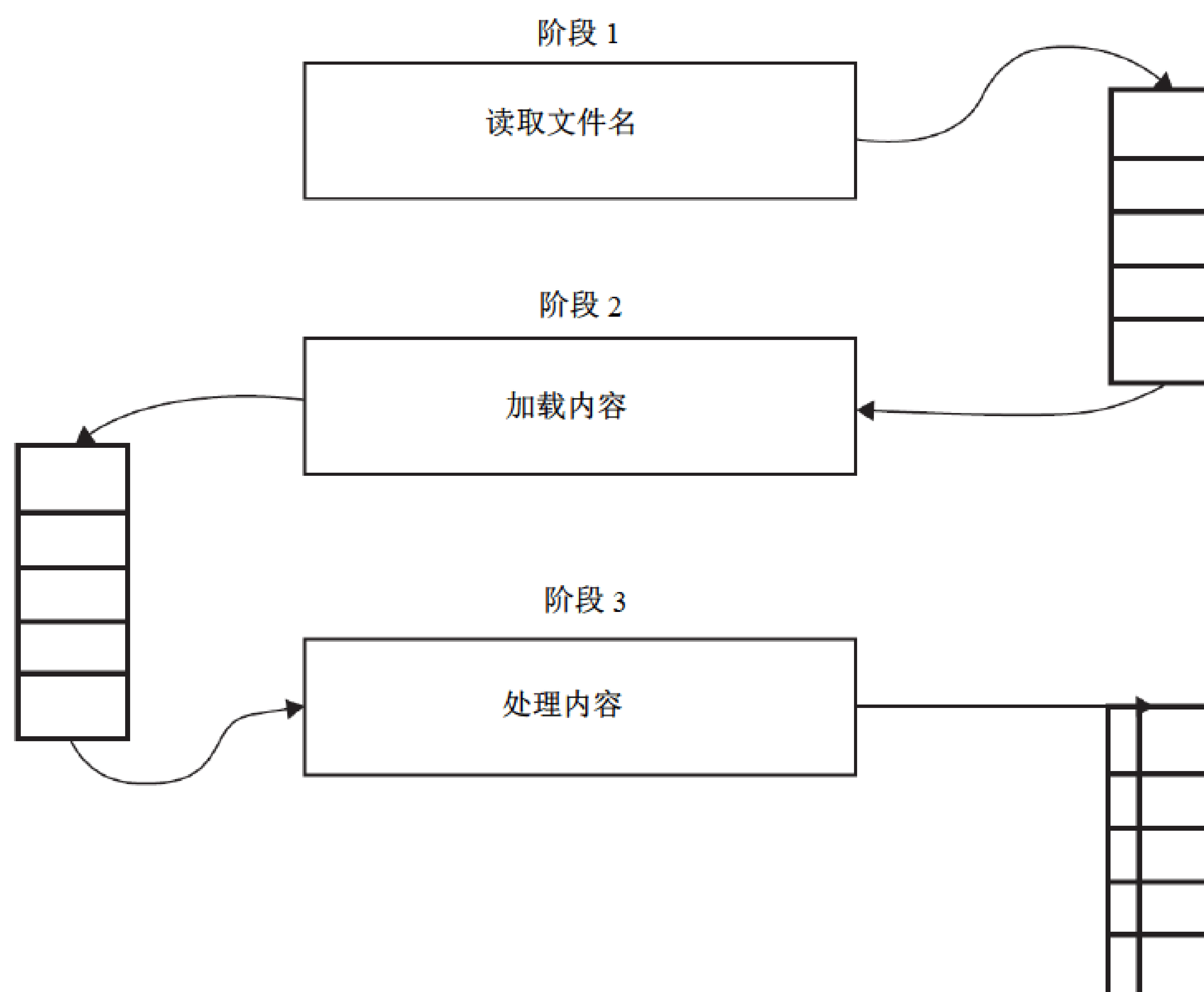



图 11-1

看看这个示例应用程序的代码可知，完整的管道是在 StartPipeline()方法中管理的。该方法实例化了集合，并把集合传递到管道的各个阶段。第 1 阶段用 ReadFilenamesAsync 处理，第 2 和第 3 阶段分别由同时运行的 LoadContentAsync 和 ProcessContentAsync 处理。但是，只有当前 3 个阶段完成后，第 4 个阶段才能启动(代码文件 PipelineSample/Program.cs)。

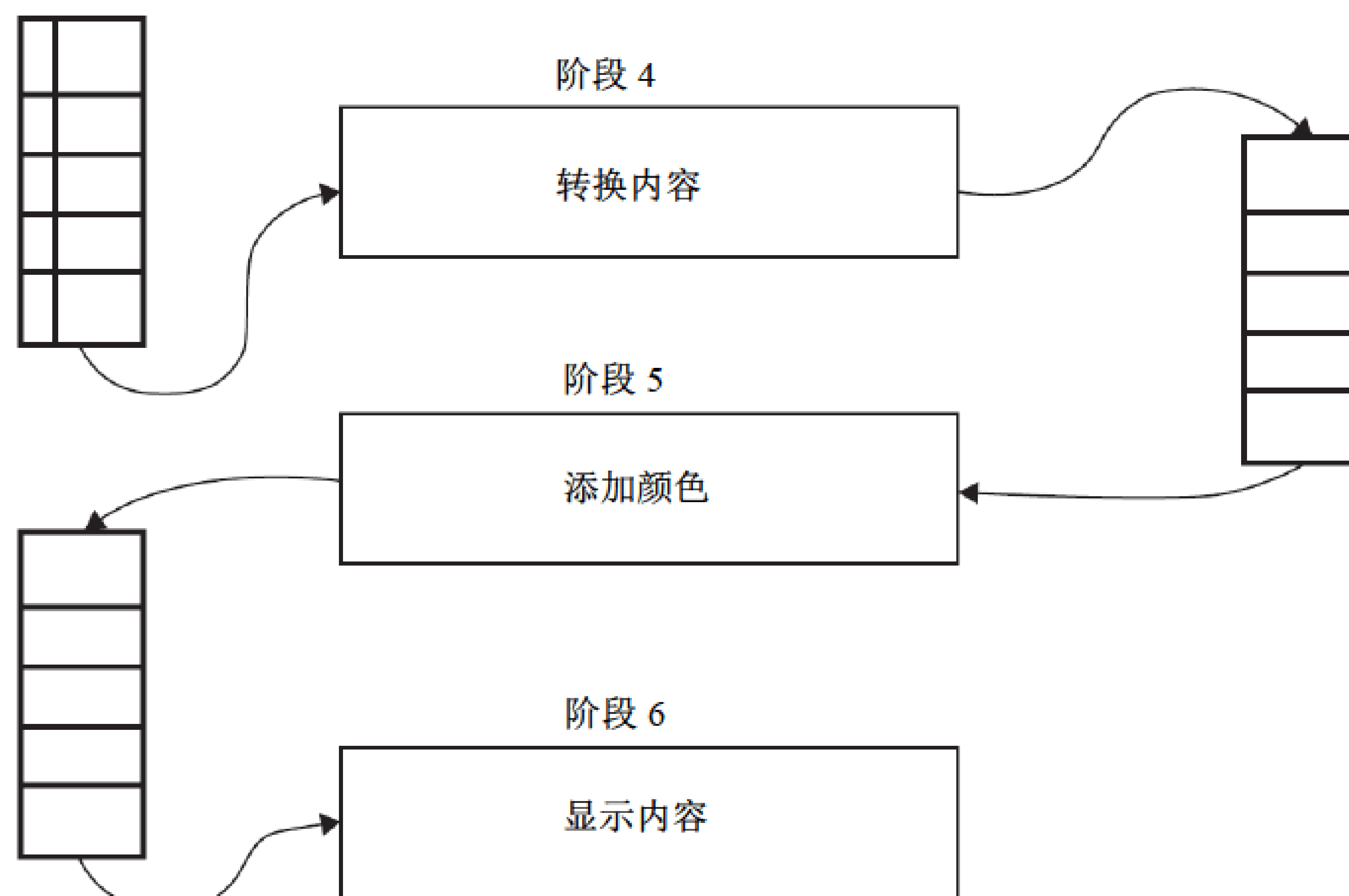



图 11-2

```
public static async Task StartPipelineAsync()
{
  var fileNames = new BlockingCollection<string>();
  var lines = new BlockingCollection<string>();
  var words = new ConcurrentDictionary<string, int>();
  var items = new BlockingCollection<Info>();
```

```
    var coloredItems = new BlockingCollection<Info>();
    Task t1 = PipelineStages.ReadFilenamesAsync(@"../../..", fileNames);
    ColoredConsole.WriteLine("started stage 1");
    Task t2 = PipelineStages.LoadContentAsync(fileNames, lines);
    ConsoleHelper.WriteLine("started stage 2");
    Task t3 = PipelineStages.ProcessContentAsync(lines, words);
    await Task.WhenAll(t1, t2, t3);
    ConsoleHelper.WriteLine("stages 1, 2, 3 completed");
    Task t4 = PipelineStages.TransferContentAsync(words, items);
    Task t5 = PipelineStages.AddColorAsync(items, coloredItems);
    Task t6 = PipelineStages.ShowContentAsync(coloredItems);
    ColoredConsole.WriteLine("stages 4, 5, 6 started");
    await Task.WhenAll(t4, t5, t6);
    ColoredConsole.WriteLine("all stages finished");
  }
```

**注意：**

这个示例应用程序使用了任务以及 async 和 await 关键字，第 15 章将介绍它们。第 21 章将详细介绍线程、任务和同步。第 22 章将讨论文件 I/O。

本例用 ColoredConsole 类向控制台写入信息。该类可以方便地改变控制台输出的颜色，并使用同步来避免返回颜色错误的输出(代码文件 PipelineSample/ConsoleHelper.cs)：

```
public static class ColoredConsole
{
  private static object syncOutput = new object();
  public static void WriteLine(string message)
  {
    lock (syncOutput)
    {
      Console.WriteLine(message);
    }
  }

  public static void WriteLine(string message, string color)
  {
    lock (syncOutput)
    {
      Console.ForegroundColor = (ConsoleColor)Enum.Parse(
        typeof(ConsoleColor), color);
      Console.WriteLine(message);
      Console.ResetColor();
    }
  }
}
```

### 11.5.2　使用 BlockingCollection

现在介绍管道的第一阶段。ReadFilenamesAsync 接收 BlockingCollection<T>为参数，在其中写入输出。该方法的实现使用枚举器来迭代指定目录及其子目录中的 C#文件。这些文件的文件名用 Add 方法添加到 BlockingCollection<T>中。完成添加文件名的操作后，调用 CompleteAdding 方法，以通知所有读取器不应再等待集合中的任何额外项(代码文件 PipelineSample/PipelineStages.cs)：

```
public static class PipelineStages
{
  public static Task ReadFilenamesAsync(string path,
    BlockingCollection<string> output)
  {
    return Task.Factory.StartNew(() =>
    {
      foreach (string filename in Directory.EnumerateFiles(path, "*.cs",
        SearchOption.AllDirectories))
      {
        output.Add(filename);
        ColoredConsole.WriteLine($"stage 1: added {filename}");
      }
      output.CompleteAdding();
    }, TaskCreationOptions.LongRunning);
  }
  //. . .
```

**注意：**

如果在写入器添加项的同时，读取器从 BlockingCollection<T>中读取，那么调用 CompleteAdding 方法是很重要的。否则，读取器会在 foreach 循环中等待更多的项被添加。

下一个阶段是读取文件并将其内容添加到另一个集合中，这由LoadContentAsync方法完成。该方法使用了输入集合传递的文件名，打开文件，然后把文件中的所有行添加到输出集合中。在foreach循环中，用输入阻塞集合调用GetConsumingEnumerable，以迭代各项。直接使用input变量而不调用GetConsumingEnumerable是可以的，但是这只会迭代当前状态的集合，而不会迭代以后添加的项。

```
public static async Task LoadContentAsync(BlockingCollection<string> input,
  BlockingCollection<string> output)
{
  foreach (var filename in input.GetConsumingEnumerable())
  {
    using (FileStream stream = File.OpenRead(filename))
    {
      var reader = new StreamReader(stream);
      string line = null;
      while ((line = await reader.ReadLineAsync()) != null)
      {
        output.Add(line);
        ColoredConsole.WriteLine($"stage 2: added {line}");
      }
    }
  }
  output.CompleteAdding();
}
```

**注意：**

如果在填充集合的同时，使用读取器读取集合，则需要使用 GetConsumingEnumerable 方法获取阻塞集合的枚举器，而不是直接迭代集合。

### 11.5.3 使用 ConcurrentDictionary

ProcessContentAsync 方法实现了第三阶段。这个方法获取输入集合中的行，然后拆分它们，将各个词筛选到输出字典中。AddOrUpdate 是 ConcurrentDictionary 类型的一个方法。如果键没有添加到字典中，第二个参数就定义应该设置的值。如果键已存在于字典中，updateValueFactory 参数就定义值的改变方式。在这种情况下，现有的值只是递增 1：

```
public static Task ProcessContentAsync(BlockingCollection<string> input,
  ConcurrentDictionary<string, int> output)
{
  return Task.Factory.StartNew(() =>
  {
    foreach (var line in input.GetConsumingEnumerable())
    {
      string[] words = line.Split(' ', ';', '\t', '{', '}', '(', ')', ':',
        ',', '"');
      foreach (var word in words.Where(w => !string.IsNullOrEmpty(w)))
      {
        output.AddOrUpdate(key: word, addValue: 1,
          updateValueFactory: (s, i) => ++i);
        ColoredConsole.WriteLine($"stage 3: added {word}");
      }
    }
  }, TaskCreationOptions.LongRunning);
}
```

运行前 3 个阶段的应用程序，得到的输出如下所示，各个阶段的操作交织在一起：

```
stage 3: added DisplayBits
stage 3: added bits2
stage 3: added Write
stage 3: added =
stage 3: added bits1.Or
stage 2: added DisplayBits(bits2);
```

```
stage 2: added Write(" and ");
stage 2: added DisplayBits(bits1);
stage 2: added WriteLine();
stage 2: added DisplayBits(bits2);
```

### 11.5.4　完成管道

在完成前 3 个阶段后，接下来的 3 个阶段也可以并行运行。TransferContentAsync 从字典中获取数据，将其转换为 Info 类型，然后放到输出 BlockingCollection<T>中(代码文件 PipelineSample/PipelineStages.cs)：

```
public static Task TransferContentAsync(
  ConcurrentDictionary<string, int> input,
  BlockingCollection<Info> output)
  {
    return Task.Factory.StartNew(() =>
    {
      foreach (var word in input.Keys)
      {
        if (input.TryGetValue(word, out int value))
        {
          var info = new Info { Word = word, Count = value };
          output.Add(info);
          ColoredConsole.WriteLine($"stage 4: added {info}");
        }
      }
      output.CompleteAdding();
    }, TaskCreationOptions.LongRunning);
  }
```

管道阶段 AddColorAsync 根据 Count 属性的值设置 Info 类型的 Color 属性：

```
public static Task AddColorAsync(BlockingCollection<Info> input,
BlockingCollection<Info> output)
{
  return Task.Factory.StartNew(() =>
  {
    foreach (var item in input.GetConsumingEnumerable())
    {
      if (item.Count > 40)
      {
        item.Color = "Red";
      }
      else if (item.Count > 20)
      {
        item.Color = "Yellow";
      }
      else
      {
        item.Color = "Green";
      }
      output.Add(item);
      ColoredConsole.WriteLine($"stage 5: added color {item.Color} to {item}");
    }
    output.CompleteAdding();
  }, TaskCreationOptions.LongRunning);
}
```

最后一个阶段用指定的颜色在控制台中输出结果：

```
public static Task ShowContentAsync(BlockingCollection<Info> input)
{
  return Task.Factory.StartNew(() =>
  {
    foreach (var item in input.GetConsumingEnumerable())
    {
      ColoredConsole.WriteLine($"stage 6: {item}", item.Color);
    }
  }, TaskCreationOptions.LongRunning);
}
```

运行应用程序，得到的结果如下所示，它是彩色的。

```
stage 6: 20 times: static
stage 6: 3 times: Count
stage 6: 2 times: t2
stage 6: 1 times: bits2[sectionD]
stage 6: 3 times: set
```

```
stage 6: 2 times: Console.ReadLine
stage 6: 3 times: started
stage 6: 1 times: builder.Remove
stage 6: 1 times: reader
stage 6: 2 times: bit4
stage 6: 1 times: ForegroundColor
stage 6: 1 times: all
all stages finished
```

## 11.6 小结

本章探讨了一些特殊的集合，如 BitArray 和 BitVector32，它们为处理带有位的集合进行了优化。

ObservableCollection < T >类不仅存储了位，列表中的项改变时，这个类还会触发事件。第 33～37 章把这个类用于 Windows 应用程序和 Xamarin 应用程序。

本章还解释了，不变的集合可以保证集合从来不会改变，因此可以很容易用于多线程应用程序。

本章的最后一部分讨论了并发集合，即可以使用一个线程填充集合，而另一个线程同时从相同的集合中检索项。

第 12 章详细讨论语言集成查询(Language Integrated Query，LINQ)。

# 第 12 章

# LINQ

**本章要点**

- 用列表在对象上执行传统查询
- 扩展方法
- LINQ 查询操作符
- 并行 LINQ
- 表达式树

**本章源代码下载地址(wrox.com)：**

打开 www.wrox.com 的 Download Code 选项卡可下载本章源代码。源代码也可以在 LINQ 目录的 https://github.com/ProfessionalCSharp/ProfessionalCSharp7 中找到。本章代码分为以下几个主要的示例文件：

- LINQ Intro
- Enumerable Sample
- Parallel LINQ
- Expression Trees

## 12.1 LINQ 概述

LINQ(Language Integrated Query，语言集成查询)在 C#编程语言中集成了查询语法，可以用相同的语法访问不同的数据源。LINQ 提供了不同数据源的抽象层，所以可以使用相同的语法。

本章介绍 LINQ 的核心原理和 C#中支持 C# LINQ Query 的语言扩展。

**注意：**

读完本章后，在数据库中使用 LINQ 的内容可查阅第 26 章，阅读本章的内容后，查询 XML 数据的内容可参见网上附加第 2 章。

在介绍 LINQ 的特性之前，本节先介绍一个简单的 LINQ 查询。C#提供了转换为方法调用的集成查询语言。本节会说明这个转换的过程，以便用户使用 LINQ 的全部功能。

### 12.1.1 列表和实体

本章的 LINQ 查询在一个包含 1950—2016 年一级方程式锦标赛的集合上进行。这些数据需要使用.NET 标准库中的类和列表来准备。

这个库使用了如下名称空间:

System

System.Collections.Generic

对于实体，定义类型 Racer。Racer 定义了几个属性和一个重载的 ToString()方法，该方法以字符串格式显示赛车手。这个类实现了 IFormattable 接口，以支持格式字符串的不同变体，这个类还实现了 IComparable<Racer>接口，它根据 Lastname 为一组赛车手排序。为了执行更高级的查询，Racer 类不仅包含单值属性，如 Firstname、Lastname、Wins、Country 和 Starts，还包含多值属性，如 Cars 和 Years。Years 属性列出了赛车手获得冠军的年份。一些赛车手曾多次获得冠军。Cars 属性用于列出赛车手在获得冠军的年份中使用的所有车型(代码文件 DataLib/Racer.cs)。

```
public class Racer: IComparable<Racer>, IFormattable
{
  public Racer(string firstName, string lastName, string country,
    int starts, int wins)
    : this(firstName, lastName, country, starts, wins, null, null) { }

  public Racer(string firstName, string lastName, string country,
    int starts, int wins, IEnumerable<int> years, IEnumerable<string> cars)
  {
    FirstName = firstName;
    LastName = lastName;
    Country = country;
    Starts = starts;
    Wins = wins;
    Years = years != null ? new List<int>(years) : new List<int>();
    Cars = cars != null ? new List<string>(cars) : new List<string>();
  }

  public string FirstName { get; }
  public string LastName { get; }
  public int Wins { get; }
  public string Country { get; }
  public int Starts { get; }
  public IEnumerable<string> Cars { get; }
  public IEnumerable<int> Years { get; }

  public override string ToString() => $"{FirstName} {LastName}";

  public int CompareTo(Racer other) => LastName.Compare(other?.LastName);

  public string ToString(string format) => ToString(format, null);

  public string ToString(string format, IFormatProvider formatProvider)
  {
    switch (format)
    {
      case null:
      case "N":
        return ToString();
      case "F":
        return FirstName;
      case "L":
        return LastName;
      case "C":
        return Country;
      case "S":
        return Starts.ToString();
      case "W":
        return Wins.ToString();
      case "A":
        return $"{FirstName} {LastName}, {Country}; starts: {Starts}, wins: {Wins}";
      default:
```

```
        throw new FormatException($"Format {format} not supported");
    }
  }
}
```

第二个实体类是 Team。这个类仅包含车队冠军的名字和获得冠军的年份。与赛车手冠军类似，针对一年中最好的车队也有一个冠军奖项(代码文件 DataLib/Team.cs)：

```
public class Team
{
  public Team(string name, params int[] years)
  {
    Name = name;
    Years = years != null ? new List<int>(years) : new List<int>();
  }
  public string Name { get; }
  public IEnumerable<int> Years { get; }
}
```

Formula1 类在 GetChampions()方法中返回一组赛车手。这个列表包含了 1950—2016 年之间的所有一级方程式冠军(代码文件 DataLib/Formula1.cs)。

```
public static class Formula1
{
  private static List<Racer> s_racers;
  public static IList<Racer> GetChampions() => s_racers ??
    (s_racers = InitalizeRacers());

  private static List<Racer> InitializeRacers =>
    new List<Racer>
    {
      new Racer("Nino", "Farina", "Italy", 33, 5, new int[] { 1950 },
        new string[] { "Alfa Romeo" }),
      new Racer("Alberto", "Ascari", "Italy", 32, 10, new int[] { 1952, 1953 },
        new string[] { "Ferrari" }),
      new Racer("Juan Manuel", "Fangio", "Argentina", 51, 24,
        new int[] { 1951, 1954, 1955, 1956, 1957 },
        new string[] { "Alfa Romeo", "Maserati", "Mercedes", "Ferrari" }),
      new Racer("Mike", "Hawthorn", "UK", 45, 3, new int[] { 1958 },
        new string[] { "Ferrari" }),
      new Racer("Phil", "Hill", "USA", 48, 3, new int[] { 1961 },
        new string[] { "Ferrari" }),
      new Racer("John", "Surtees", "UK", 111, 6, new int[] { 1964 },
        new string[] { "Ferrari" }),
      new Racer("Jim", "Clark", "UK", 72, 25, new int[] { 1963, 1965 },
        new string[] { "Lotus" }),
      new Racer("Jack", "Brabham", "Australia", 125, 14,
        new int[] { 1959, 1960, 1966 }, new string[] { "Cooper", "Brabham" }),
      new Racer("Denny", "Hulme", "New Zealand", 112, 8, new int[] { 1967 },
        new string[] { "Brabham" }),
      new Racer("Graham", "Hill", "UK", 176, 14, new int[] { 1962, 1968 },
        new string[] { "BRM", "Lotus" }),
      new Racer("Jochen", "Rindt", "Austria", 60, 6, new int[] { 1970 },
        new string[] { "Lotus" }),
      new Racer("Jackie", "Stewart", "UK", 99, 27,
        new int[] { 1969, 1971, 1973 }, new string[] { "Matra", "Tyrrell" }),
      //...
  }
  //...
}
```

对于后面在多个列表中执行的查询，GetConstructorChampions()方法返回所有的车队冠军的列表。车队冠军是从 1958 年开始设立的。

```
private static List<Team> s_teams;
public static IList<Team> GetContructorChampions()
{
  if (s_teams == null)
  {
    s_teams = new List<Team>()
    {
      new Team("Vanwall", 1958),
      new Team("Cooper", 1959, 1960),
      new Team("Ferrari", 1961, 1964, 1975, 1976, 1977, 1979, 1982, 1983, 1999,
        2000, 2001, 2002, 2003, 2004, 2007, 2008),
      new Team("BRM", 1962),
      new Team("Lotus", 1963, 1965, 1968, 1970, 1972, 1973, 1978),
```

```
        new Team("Brabham", 1966, 1967),
        new Team("Matra", 1969),
        new Team("Tyrrell", 1971),
        new Team("McLaren", 1974, 1984, 1985, 1988, 1989, 1990, 1991, 1998),
        new Team("Williams", 1980, 1981, 1986, 1987, 1992, 1993, 1994, 1996,
          1997),
        new Team("Benetton", 1995),
        new Team("Renault", 2005, 2006),
        new Team("Brawn GP", 2009),
        new Team("Red Bull Racing", 2010, 2011, 2012, 1013),
        new Team("Mercedes", 2014, 2015, 2016, 2017)
      };
    }
    return s_teams;
  }
```

### 12.1.2 LINQ 查询

演示 LINQ 的示例应用程序是一个控制台应用程序，使用了如下名称空间：

System

System.Collections.Generic

System.Linq

在以前创建的库中，使用这些准备好的列表和实体，进行 LINQ 查询，例如，查询来自巴西的所有世界冠军，并按照夺冠次数排序。为此可以使用 List<T>类的方法，如 FindAll()和 Sort()方法。而使用 LINQ 的语法非常简单(代码文件 LINQIntro/Program.cs)：

```
static void LINQQuery()
{
  var query = from r in Formula1.GetChampions()
              where r.Country == "Brazil"
              orderby r.Wins descending
              select r;

  foreach (Racer r in query)
  {
    Console.WriteLine($"{r:A}");
  }
}
```

这个查询的结果显示了来自巴西的所有世界冠军，并排好序：

```
Ayrton Senna, Brazil; starts: 161, wins: 41
Nelson Piquet, Brazil; starts: 204, wins: 23
Emerson Fittipaldi, Brazil; starts: 143, wins: 14
```

表达式

```
from r in Formula1.GetChampions()
where r.Country == "Brazil"
orderby r.Wins descending
select r;
```

是一个 LINQ 查询。子句 from、where、orderby、descending 和 select 都是这个查询中预定义的关键字。

查询表达式必须以 from 子句开头，以 select 或 group 子句结束。在这两个子句之间，可以使用 where、orderby、join、let 和其他 from 子句。

**注意：**

变量 query 只指定了 LINQ 查询。该查询不是通过这个赋值语句执行的，只要使用 foreach 循环访问查询，该查询就会执行。

### 12.1.3 扩展方法

编译器会转换 LINQ 查询，以调用方法而不是 LINQ 查询。LINQ 为 IEnumerable<T>接口提供了各种扩展方法，以便用户在实现了该接口的任意集合上使用 LINQ 查询。扩展方法在静态类中声明，定义为一个静态方

法，其中第一个参数定义了它扩展的类型。

扩展方法可以将方法写入最初没有提供该方法的类中。还可以把方法添加到实现某个特定接口的任何类中，这样多个类就可以使用相同的实现代码。

例如，String 类没有 Foo()方法。String 类是密封的，所以不能从这个类中继承。但可以创建一个扩展方法，如下所示：

```
public static class StringExtension
{
  public static void Foo(this string s)
  {
    Console.WriteLine($"Foo invoked for {s}");
  }
}
```

Foo()方法扩展了 String 类，因为它的第一个参数定义为 String 类型。为了区分扩展方法和一般的静态方法，扩展方法还需要对第一个参数使用 this 关键字。现在就可以使用带 string 类型的 Foo()方法了：

```
string s = "Hello";
s.Foo();
```

结果在控制台上显示“Foo invoked for Hello”，因为 Hello 是传递给 Foo()方法的字符串。

也许这看起来违反了面向对象的规则，因为给一个类型定义了新方法，但没有改变该类型或派生自它的类型。但实际上并非如此。扩展方法不能访问它扩展的类型的私有成员。调用扩展方法只是调用静态方法的一种新语法。对于字符串，可以用如下方式调用 Foo()方法，获得相同的结果：

```
string s = "Hello";
StringExtension.Foo(s);
```

要调用静态方法，应在类名的后面加上方法名。扩展方法是调用静态方法的另一种方式。不必提供定义了静态方法的类名，相反，编译器调用静态方法是因为它带的参数类型。只需要导入包含该类的名称空间，就可以将 Foo()扩展方法放在 String 类的作用域中。

定义 LINQ 扩展方法的一个类是 System.Linq 名称空间中的 Enumerable。只需要导入这个名称空间，就可以打开这个类的扩展方法的作用域。下面列出了 Where()扩展方法的实现代码。Where()扩展方法的第一个参数包含了 this 关键字，其类型是 IEnumerable<T>。这样，Where()方法就可以用于实现 IEnumerable<T>的每个类型。例如，数组和 List<T>类实现了 IEnumerable<T>接口。第二个参数是一个 Func<T,bool>委托，它引用了一个返回布尔值、参数类型为 T 的方法。这个谓词在实现代码中调用，检查 IEnumerable<T>源中的项是否应放在目标集合中。如果委托引用了该方法，yield return 语句就将源中的项返回给目标。

```
public static IEnumerable<TSource> Where<TSource>(
  this IEnumerable<TSource> source,
  Func<TSource, bool> predicate)
{
  foreach (TSource item in source)
  {
    if (predicate(item))
      yield return item;
  }
}
```

因为 Where()作为一个泛型方法实现，所以它可以用于包含在集合中的任意类型。实现了 IEnumerable<T>接口的任意集合都支持它。

**注意：**

*这里的扩展方法在 System.Core 程序集的 System.Linq 名称空间中定义。*

现在就可以使用Enumerable类中的扩展方法Where()、OrderByDescending()和Select()。这些方法都返回IEnumerable<TSource>，所以可以使用前面的结果依次调用这些方法。通过扩展方法的参数，使用定义了委托参数的实现代码的匿名方法(代码文件LINQIntro/Program.cs)。

```
static void ExtensionMethods()
{
```

```
var champions = new List<Racer>(Formula1.GetChampions());
IEnumerable<Racer> brazilChampions =
  champions.Where(r =< r.Country == "Brazil")
    .OrderByDescending(r =< r.Wins)
    .Select(r =< r);

foreach (Racer r in brazilChampions)
{
  Console.WriteLine($"{r:A}");
}
```

### 12.1.4 推迟查询的执行

在运行期间定义查询表达式时，查询就不会运行。查询会在迭代数据项时运行。

再看看扩展方法 Where()。它使用 yield return 语句返回谓词为 true 的元素。因为使用了 yield return 语句，所以编译器会创建一个枚举器，在访问枚举中的项后，就返回它们。

```
public static IEnumerable<T> Where<T>(this IEnumerable<T> source,
  Func<T, bool> predicate)
  {
    foreach (T item in source)
    {
      if (predicate(item))
      {
        yield return item;
      }
    }
  }
}
```

这是一个非常有趣也非常重要的结果。在下面的例子中，创建了 String 元素的一个集合，用名称填充它。接着定义一个查询，从集合中找出以字母 J 开头的所有名称。集合也应是排好序的。在定义查询时，不会进行迭代。相反，迭代在 foreach 语句中进行，在其中迭代所有的项。集合中只有一个元素 Juan 满足 where 表达式的要求，即以字母 J 开头。迭代完成后，将 Juan 写入控制台。之后在集合中添加 4 个新名称，再次进行迭代(代码文件 LINQIntro/Program.cs)。

```
static void DeferredQuery()
{
  var names = new List<string> { "Nino", "Alberto", "Juan", "Mike", "Phil" };
  var namesWithJ = from n in names
                   where n.StartsWith("J")
                   orderby n
                   select n;

  Console.WriteLine("First iteration");
  foreach (string name in namesWithJ)
  {
    Console.WriteLine(name);
  }
  Console.WriteLine();

  names.Add("John");
  names.Add("Jim");
  names.Add("Jack");
  names.Add("Denny");
  Console.WriteLine("Second iteration");

  foreach (string name in namesWithJ)
  {
    Console.WriteLine(name);
  }
```

因为迭代在查询定义时不会进行，而是在执行每个 foreach 语句时进行，所以可以看到其中的变化，如应用程序的结果所示：

```
First iteration
Juan
Second iteration
Jack
Jim
John
Juan
```

当然，还必须注意，每次在迭代中使用查询时，都会调用扩展方法。大多数情况下，这是非常有效的，因为我们可以检测出源数据中的变化。但在一些情况下，这是不可行的。调用扩展方法 ToArray()、ToList()等可以改变这个操作。在示例中，ToList 遍历集合，返回一个实现了 IList<string>的集合。之后对返回的列表遍历两次，在两次迭代之间，数据源得到了新名称。

```
var names = new List<string> { "Nino", "Alberto", "Juan", "Mike", "Phil" };
var namesWithJ = (from n in names
                  where n.StartsWith("J")
                  orderby n
                  select n).ToList();

Console.WriteLine("First iteration");
foreach (string name in namesWithJ)
{
  Console.WriteLine(name);
}
Console.WriteLine();

names.Add("John");
names.Add("Jim");
names.Add("Jack");
names.Add("Denny");
Console.WriteLine("Second iteration");
foreach (string name in namesWithJ)
{
  Console.WriteLine(name);
}
```

在结果中可以看到，在两次迭代之间输出保持不变，但集合中的值改变了：

```
First iteration
Juan
Second iteration
Juan
```

# 12.2　标准的查询操作符

Where、OrderByDescending 和 Select 只是 LINQ 定义的几个查询操作符。LINQ 查询为最常用的操作符定义了一个声明语法。还有许多查询操作符可用于 Enumerable 类。

表 12-1 列出了 Enumerable 类定义的标准查询操作符。

表 12-1

| 标准查询操作符 | 说　明 |
| --- | --- |
| Where<br>OfType<TResult> | 筛选操作符定义了返回元素的条件。在 Where 查询操作符中可以使用谓词，例如，lambda 表达式定义的谓词，来返回布尔值。OfType<TResult>根据类型筛选元素，只返回 TResult 类型的元素 |
| Select<br>SelectMany | 投射操作符用于把对象转换为另一个类型的新对象。Select 和 SelectMany 定义了根据选择器函数选择结果值的投射 |
| OrderBy<br>ThenBy<br>OrderByDescending<br>ThenByDescending<br>Reverse | 排序操作符改变所返回的元素的顺序。OrderBy 按升序排序，OrderByDescending 按降序排序。如果第一次排序的结果很类似，就可以使用 ThenBy 和 ThenByDescending 操作符进行第二次排序。Reverse 反转集合中元素的顺序 |
| Join<br>GroupJoin | 连接操作符用于合并不直接相关的集合。使用 Join 操作符，可以根据键选择器函数连接两个集合，这类似于 SQL 中的 JOIN。GroupJoin 操作符连接两个集合，组合其结果 |
| GroupBy<br>ToLookup | 组合操作符把数据放在组中。GroupBy 操作符组合有公共键的元素。ToLookup 通过创建一个一对多字典，来组合元素 |

(续表)

| 标准查询操作符 | 说　明 |
| --- | --- |
| Any<br>All<br>Contains | 如果元素序列满足指定的条件，限定符操作符就返回布尔值。Any、All 和 Contains 都是限定符操作符。Any 确定集合中是否有满足谓词函数的元素；All 确定集合中的所有元素是否都满足谓词函数；Contains 检查某个元素是否在集合中 |
| Take<br>Skip<br>TakeWhile<br>SkipWhile | 分区操作符返回集合的一个子集。Take、Skip、TakeWhile 和 SkipWhile 都是分区操作符。使用它们可以得到部分结果。使用 Take 必须指定要从集合中提取的元素个数；Skip 跳过指定的元素个数，提取其他元素；TakeWhile 提取条件为真的元素，SkipWhile 跳过条件为真的元素 |
| Distinct<br>Union<br>Intersect<br>Except<br>Zip | Set 操作符返回一个集合。Distinct 从集合中删除重复的元素。除了 Distinct 之外，其他 Set 操作符都需要两个集合。Union 返回出现在其中一个集合中的唯一元素。Intersect 返回两个集合中都有的元素。Except 返回只出现在一个集合中的元素。Zip 把两个集合合并为一个 |
| First<br>FirstOrDefault<br>Last<br>LastOrDefault<br>ElementAt<br>ElementAtOrDefault<br>Single<br>SingleOrDefault | 这些元素操作符仅返回一个元素。First 返回第一个满足条件的元素。FirstOrDefault 类似于 First，但如果没有找到满足条件的元素，就返回类型的默认值。Last 返回最后一个满足条件的元素。ElementAt 指定了要返回的元素的位置。Single 只返回一个满足条件的元素。如果有多个元素都满足条件，就抛出一个异常。所有的 *XX*OrDefault 方法都类似于以相同前缀开头的方法，但如果没有找到该元素，它们就返回类型的默认值 |
| Count<br>Sum<br>Min<br>Max<br>Average<br>Aggregate | 聚合操作符计算集合的一个值。利用这些聚合操作符，可以计算所有值的总和、所有元素的个数、值最大和最小的元素，以及平均值等 |
| ToArray<br>AsEnumerable<br>ToList<br>ToDictionary<br>Cast<TResult> | 这些转换操作符将集合转换为数组：IEnumerable、IList、IDictionary 等。Cast 方法把集合的每个元素类型转换为泛型参数类型 |
| Empty<br>Range<br>Repeat | 这些生成操作符返回一个新集合。使用 Empty 时集合是空的；Range 返回一系列数字；Repeat 返回一个始终重复一个值的集合 |

下面是使用这些操作符的一些例子。

### 12.2.1 筛选

下面介绍一些查询的示例。示例应用程序是一个控制台应用程序，使用了如下名称空间：

System
System.Collections.Generic
System.Linq

这个带有下载代码的示例应用程序提供了上述每个不同特性的传递命令行参数。在 Properties 页面的 Debug 部分，可以根据需要配置命令行参数，以运行应用程序的不同部分。在命令行中，可以使用.NET Core 命令行实用程序以以下方式调用命令：

```
dotnet run -- -f
```

其中将参数-f 传递给应用程序。

使用 where 子句，可以合并多个表达式。例如，找出赢得至少 15 场比赛的巴西和奥地利赛车手。传递给 where 子句的表达式的结果类型应是布尔类型(代码文件 EnumerableSample/FilteringSamples.cs)：

```
static void Filtering()
{
  var racers = from r in Formula1.GetChampions()
               where r.Wins > 15 &&
               (r.Country == "Brazil" || r.Country == "Austria")
               select r;

  foreach (var r in racers)
  {
    Console.WriteLine($"{r:A}");
  }
}
```

用这个 LINQ 查询启动程序，会返回 Niki Lauda、Nelson Piquet 和 Ayrton Senna，如下：

```
Niki Lauda, Austria, Starts: 173, Wins: 25
Nelson Piquet, Brazil, Starts: 204, Wins: 23
Ayrton Senna, Brazil, Starts: 161, Wins: 41
```

并不是所有的查询都可以用 LINQ 查询语法完成。也不是所有的扩展方法都映射到 LINQ 查询子句上。高级查询需要使用扩展方法。为了更好地理解带扩展方法的复杂查询，最好看看简单的查询是如何映射的。使用扩展方法 Where()和 Select()，会生成与前面 LINQ 查询非常类似的结果(代码文件 EnumerableSample/FilteringSamples.cs)：

```
static void FilteringWithMethods()
{
  var racers = Formula1.GetChampions()
    .Where(r => r.Wins > 15 &&
      (r.Country == "Brazil" || r.Country == "Austria"))
      .Select(r => r);
  //...
}
```

### 12.2.2 用索引筛选

不能使用 LINQ 查询的一个例子是 Where()方法的重载。在 Where()方法的重载中，可以传递第二个参数——索引。索引是筛选器返回的每个结果的计数器。可以在表达式中使用这个索引，执行基于索引的计算。下面的代码由 Where()扩展方法调用，它使用索引返回姓氏以 A 开头、索引为偶数的赛车手(代码文件 EnumerableSample/FilteringSamples.cs)：

```
static void FilteringWithIndex()
{
  var racers = Formula1.GetChampions()
    .Where((r, index) => r.LastName.StartsWith("A") && index % 2 != 0);

  foreach (var r in racers)
  {
    Console.WriteLine($"{r:A}");
  }
}
```

姓氏以 A 开头的所有赛车手有 Alberto Ascari、Mario Andretti 和 Fernando Alonso。因为 Mario Andretti 的索引是奇数，所以他不在结果中：

```
Alberto Ascari, Italy; starts: 32, wins: 10
Fernando Alonso, Spain; starts: 279, wins: 33
```

### 12.2.3 类型筛选

为了进行基于类型的筛选，可以使用 OfType()扩展方法。这里数组数据包含 string 和 int 对象。使用 OfType()扩展方法，把 string 类传送给泛型参数，就从集合中仅返回字符串(代码文件 EnumerableSample/FilteringSamples.cs)：

```
static void TypeFiltering()
{
  object[] data = { "one", 2, 3, "four", "five", 6 };
  var query = data.OfType<string>();

  foreach (var s in query)
  {
    Console.WriteLine(s);
  }
}
```

运行这段代码，就会显示字符串 one、four 和 five。

```
one
four
five
```

### 12.2.4 复合的 from 子句

如果需要根据对象的一个成员进行筛选，而该成员本身是一个系列，就可以使用复合的 from 子句。Racer 类定义了一个属性 Cars，其中 Cars 是一个字符串数组。要筛选驾驶法拉利的所有冠军，可以使用如下所示的 LINQ 查询。第一个 from 子句访问从 Formula1.GetChampions()方法返回的 Racer 对象，第二个 from 子句访问 Racer 类的 Cars 属性，以返回所有 string 类型的赛车。接着在 where 子句中使用这些赛车筛选驾驶法拉利的所有冠军(代码文件 EnumerableSample/CompoundFromSamples.cs)。

```
static void CompoundFrom()
{
  var ferrariDrivers = from r in Formula1.GetChampions()
                       from c in r.Cars
                       where c == "Ferrari"
                       orderby r.LastName
                       select r.FirstName + " " + r.LastName;
  //...
}
```

这个查询的结果显示了驾驶法拉利的所有一级方程式冠军：

```
Alberto Ascari
Juan Manuel Fangio
Mike Hawthorn
Phil Hill
Niki Lauda
Kimi Räikkönen
Jody Scheckter
Michael Schumacher
John Surtees
```

C#编译器把复合的 from 子句和 LINQ 查询转换为 SelectMany()扩展方法。SelectMany()方法可用于迭代序列的序列。示例中 SelectMany()方法的重载版本如下所示：

```
public static IEnumerable<TResult> SelectMany<TSource, TCollection, TResult> (
  this IEnumerable<TSource> source,
  Func<TSource,
  IEnumerable<TCollection>> collectionSelector,
  Func<TSource, TCollection, TResult> resultSelector);
```

第一个参数是隐式参数，它从 GetChampions()方法中接收 Racer 对象序列。第二个参数是 collectionSelector 委托，其中定义了内部序列。在 lambda 表达式 r => r.Cars 中，应返回赛车集合。第三个参数是一个委托，现在为每个赛车调用该委托，接收 Racer 和 Car 对象。lambda 表达式创建了一个匿名类型，它有 Racer 和 Car 属性。这个 SelectMany()方法的结果是摊平了赛车手和赛车的层次结构，为每辆赛车返回匿名类型的一个新对象集合。

这个新集合传递给 Where()方法，筛选出驾驶法拉利的赛车手。最后，调用 OrderBy()和 Select()方法(代码

文件 EnumerableSample/CompoundFromSamples.cs)：

```
static void CompoundFromWithMethods()
{
  var ferrariDrivers = Formula1.GetChampions()
    .SelectMany(r => r.Cars, (r, c) => new { Racer = r, Car = c })
    .Where(r => r.Car == "Ferrari")
    .OrderBy(r => r.Racer.LastName)
    .Select(r => r.Racer.FirstName + " " + r.Racer.LastName);
  //...
}
```

把 SelectMany()泛型方法解析为这里使用的类型，所解析的类型如下所示。在这个例子中，数据源是 Racer 类型，所筛选的集合是一个 string 数组，当然所返回的匿名类型的名称是未知的，这里显示为 TResult：

```
public static IEnumerable<TResult> SelectMany<Racer, string, TResult> (
  this IEnumerable<Racer> source,
  Func<Racer, IEnumerable<string>> collectionSelector,
  Func<Racer, string, TResult> resultSelector);
```

因为查询仅从 LINQ 查询转换为扩展方法，所以结果与前面的相同。

### 12.2.5　排序

要对序列排序，前面使用了 orderby 子句。下面复习一下前面使用的例子，但这里使用 orderby descending 子句。其中赛车手按照赢得比赛的次数进行降序排序，赢得比赛的次数用关键字选择器指定(代码文件 EnumerableSample/SortingSamples.cs)：

```
static void SortDescending()
{
  var racers = from r in Formula1.GetChampions()
               where r.Country == "Brazil"
               orderby r.Wins descending
               select r;
  //...
}
```

orderby 子句解析为 OrderBy()方法，orderby descending 子句解析为 OrderByDescending()方法(代码文件 EnumerableSample/SortingSamples.cs)：

```
static void SortDescendingWithMethods()
{
  var racers = Formula1.GetChampions()
    .Where(r => r.Country == "Brazil")
    .OrderByDescending(r => r.Wins)
    .Select(r => r);
  //...
}
```

OrderBy()和 OrderByDescending()方法返回 IOrderEnumerable<TSource>。这个接口派生自 IEnumerable<TSource> 接口，但包含一个额外的方法 CreateOrderedEnumerable<TSource>()。这个方法用于进一步给序列排序。如果根据关键字选择器来排序，其中有两项相同，就可以使用 ThenBy()和 ThenByDescending ()方法继续排序。这两个方法需要 IOrderEnumerable<TSource>接口才能工作，但也返回这个接口。所以，可以添加任意多个 ThenBy()和 ThenByDescending()方法，对集合排序。

使用 LINQ 查询时，只需要把所有用于排序的不同关键字(用逗号分隔开)添加到 orderby 子句中。在下例中，所有的赛车手先按照国家排序，再按照姓氏排序，最后按照名字排序。添加到 LINQ 查询结果中的 Take()扩展方法用于返回前 10 个结果：

```
static void SortMultiple()
{
  var racers = (from r in Formula1.GetChampions()
                orderby r.Country, r.LastName, r.FirstName
                select r).Take(10);
  //...
}
```

排序后的结果如下：

```
Argentina: Fangio, Juan Manuel
Australia: Brabham, Jack
Australia: Jones, Alan
Austria: Lauda, Niki
Austria: Rindt, Jochen
Brazil: Fittipaldi, Emerson
Brazil: Piquet, Nelson
Brazil: Senna, Ayrton
Canada: Villeneuve, Jacques
Finland: Hakkinen, Mika
```

使用 OrderBy()和 ThenBy()扩展方法可以执行相同的操作(代码文件 EnumerableSample/SortingSamples.cs)：

```
static void SortMultipleWithMethods()
{
  var racers = Formula1.GetChampions()
    .OrderBy(r => r.Country)
    .ThenBy(r => r.LastName)
    .ThenBy(r => r.FirstName)
    .Take(10);
  //...
}
```

### 12.2.6 分组

要根据一个关键字值对查询结果分组，可以使用 group 子句。现在一级方程式冠军应按照国家分组，并列出一个国家的冠军数。子句 group r by r.Country into g 根据 Country 属性组合所有的赛车手，并定义一个新的标识符 g，它以后用于访问分组的结果信息。group 子句的结果根据应用到分组结果上的扩展方法 Count()来排序，如果冠军数相同，就根据关键字来排序，该关键字是国家，因为这是分组所使用的关键字。where 子句根据至少有两项的分组来筛选结果，select 子句创建一个带 Country 和 Count 属性的匿名类型(代码文件 EnumerableSample/GroupingSamples.cs)。

```
static void Grouping()
{
  var countries = from r in Formula1.GetChampions()
                  group r by r.Country into g
                  orderby g.Count() descending, g.Key
                  where g.Count() >= 2
                  select new
                  {
                    Country = g.Key,
                    Count = g.Count()
                  };

  foreach (var item in countries)
  {
    Console.WriteLine($"{item.Country, -10} {item.Count}");
  }
}
```

结果显示了带 Country 和 Count 属性的对象集合：

```
UK          10
Brazil      3
Finland     3
Germany     3
Australia   2
Austria     2
Italy       2
USA         2
```

要用扩展方法执行相同的操作，应把 groupby 子句解析为 GroupBy()方法。在 GroupBy()方法的声明中，注意它返回实现了 IGrouping 接口的枚举对象。IGrouping 接口定义了 Key 属性，所以在定义了对这个方法的调用后，可以访问分组的关键字：

```
public static IEnumerable<IGrouping<TKey, TSource>> GroupBy<TSource, TKey>(
  this IEnumerable<TSource> source, Func<TSource, TKey> keySelector);
```

把子句 group r by r.Country into g 解析为 GroupBy(r => r.Country)，返回分组序列。分组序列首先用

OrderByDescending()方法排序，再用 ThenBy()方法排序。接着调用 Where()和 Select()方法(代码文件 EnumerableSample/GroupingSamples.cs)。

```
static void GroupingWithMethods()
{
  var countries = Formula1.GetChampions()
    .GroupBy(r => r.Country)
    .OrderByDescending(g => g.Count())
    .ThenBy(g => g.Key)
    .Where(g => g.Count() >= 2)
    .Select(g => new
    {
      Country = g.Key,
      Count = g.Count()
    });
```

### 12.2.7　LINQ 查询中的变量

在为分组编写的 LINQ 查询中，Count 方法调用了多次。使用 let 子句可以改变这种方式。let 允许在 LINQ 查询中定义变量(代码文件 EnumerableSample/GroupingSamples.cs)：

```
static void GroupingWithVariables()
{
  var countries = from r in Formula1.GetChampions()
                  group r by r.Country into g
                  let count = g.Count()
                  orderby count descending, g.Key
                  where count >= 2
                  select new
                  {
                    Country = g.Key,
                    Count = count
                  };
  //...
}
```

使用方法语法，Count 方法也调用了多次。为了定义传递给下一个方法的额外数据 (let 子句执行的操作)，可以使用 Select 方法来创建匿名类型。这里创建了一个带 Group 和 Count 属性的匿名类型。带有这些属性的一组项传递给 OrderByDescending 方法，基于匿名类型的 Count 属性排序：

```
static void GroupingWithAnonymousTypes()
{
  var countries = Formula1.GetChampions()
    .GroupBy(r => r.Country)
    .Select(g => new { Group = g, Count = g.Count() })
    .OrderByDescending(g => g.Count)
    .ThenBy(g => g.Group.Key)
    .Where(g => g.Count >= 2)
    .Select(g => new
    {
      Country = g.Group.Key,
      Count = g.Count
    });
  //...
}
```

应考虑根据 let 子句或 Select 方法创建的临时对象的数量。查询大列表时，创建的大量对象需要以后进行垃圾收集，这可能对性能产生巨大影响。

### 12.2.8　对嵌套的对象分组

如果分组的对象应包含嵌套的序列，就可以改变 select 子句创建的匿名类型。在下面的例子中，所返回的国家不仅应包含国家名和赛车手数量这两个属性，还应包含赛车手的名序列。这个序列用一个赋予 Racers 属性的 from/in 内部子句指定，内部的 from 子句使用分组标识符 g 获得该分组中的所有赛车手，用姓氏对它们排序，再根据姓名创建一个新字符串(代码文件 EnumerableSample/GroupingSamples.cs)：

```
static void GroupingAndNestedObjects()
{
  var countries = from r in Formula1.GetChampions()
```

```
                        group r by r.Country into g
                        let count = g.Count()
                        orderby count descending, g.Key
                        where count >= 2
                        select new
                        {
                          Country = g.Key,
                          Count = count,
                          Racers = from r1 in g
                                   orderby r1.LastName
                                   select r1.FirstName + " " + r1.LastName
                        };

  foreach (var item in countries)
  {
    Console.WriteLine($"{item.Country, -10} {item.Count}");
    foreach (var name in item.Racers)
    {
      Console.Write($"{name}; ");
    }
    Console.WriteLine();
  }
}
```

使用扩展方法，内部 Racer 对象是使用 IGrouping 类型的 group 变量 g 创建的，其中 Key 属性是分组的键(本例中的国家)，可以使用 Group 属性访问组的项：

```
static void GroupingAndNestedObjectsWithMethods()
{
  var countries = Formula1.GetChampions()
    .GroupBy(r => r.Country)
    .Select(g => new
    {
      Group = g,
      Key = g.Key,
      Count = g.Count()
    })
    .OrderByDescending(g => g.Count)
    .ThenBy(g => g.Key)
    .Where(g => g.Count >= 2)
    .Select(g => new
    {
      Country = g.Key,
      Count = g.Count,
      Racers = g.Group.OrderBy(r => r.LastName)
        .Select(r => r.FirstName + " " + r.LastName)
    });
  //...
}
```

结果应列出某个国家的所有冠军：

```
UK         10
Jenson Button; Jim Clark; Lewis Hamilton; Mike Hawthorn; Graham Hill;
Damon Hill; James Hunt; Nigel Mansell; Jackie Stewart; John Surtees;
Brazil     3
Emerson Fittipaldi; Nelson Piquet; Ayrton Senna;
Finland    3
Mika Hakkinen; Kimi Raikkonen; Keke Rosberg;
Germany    3
Nico Rosberg; Michael Schumacher; Sebastian Vettel;
Australia  2
Jack Brabham; Alan Jones;
Austria    2
Niki Lauda; Jochen Rindt;
Italy      2
Alberto Ascari; Nino Farina;
USA        2
Mario Andretti; Phil Hill;
```

### 12.2.9 内连接

使用 join 子句可以根据特定的条件合并两个数据源，但之前要获得两个要连接的列表。在一级方程式比赛中，有赛车手冠军和车队冠军。赛车手从 GetChampions()方法中返回，车队从 GetConstructorChampions()

方法中返回。现在要获得一个年份列表，列出每年的赛车手冠军和车队冠军。

为此，先定义两个查询，用于查询赛车手和车队(代码文件 EnumerableSample/JoinSamples.cs)：

```
static void InnerJoin()
{
  var racers = from r in Formula1.GetChampions()
               from y in r.Years
               select new
               {
                 Year = y,
                 Name = r.FirstName + " " + r.LastName
               };

  var teams = from t in Formula1.GetContructorChampions()
              from y in t.Years
              select new
              {
                Year = y,
                Name = t.Name
              };
  //...
}
```

有了这两个查询，再通过 join 子句，根据赛车手获得冠军的年份和车队获得冠军的年份进行连接。select 子句定义了一个新的匿名类型，它包含 Year、Racer 和 Team 属性。

```
var racersAndTeams = (from r in racers
                      join t in teams on r.Year equals t.Year
                      select new
                      {
                        r.Year,
                        Champion = r.Name,
                        Constructor = t.Name
                      }).Take(10);
Console.WriteLine("Year World Champion\t Constructor Title");

foreach (var item in racersAndTeams)
{
  Console.WriteLine($"{item.Year}: {item.Champion,-20} {item.Constructor}");
}
```

当然，也可以把它们合并为一个 LINQ 查询，但这只是一种个人喜好的问题：

```
var racersAndTeams =
  (from r in
   from r1 in Formula1.GetChampions()
   from yr in r1.Years
   select new
   {
     Year = yr,
     Name = r1.FirstName + " " + r1.LastName
   }
   join t in
     from t1 in Formula1.GetContructorChampions()
     from yt in t1.Years
     select new
     {
       Year = yt,
       Name = t1.Name
     }
   on r.Year equals t.Year
   orderby t.Year
   select new
   {
     Year = r.Year,
     Racer = r.Name,
     Team = t.Name
   }).Take(10);
```

使用扩展方法可以加入赛车手和车队，具体操作是调用 Join 方法，通过第一个参数传递车队，把他们与赛车手连接起来，指定外部和内部集合的关键字选择器，并通过最后一个参数定义结果选择器(代码文件 EnumerableSample/JoinSamples.cs)：

```
static void InnerJoinWithMethods()
{
```

```
  var racers = Formula1.GetChampions()
    .SelectMany(r => r.Years, (r1, year) =>
    new
    {
      Year = year,
      Name = $"{r1.FirstName} {r1.LastName}"
    });

  var teams = Formula1.GetConstructorChampions()
    .SelectMany(t => t.Years, (t, year) =>
    new
    {
      Year = year,
      Name = t.Name
    });

  var racersAndTeams = racers.Join(
    teams,
    r => r.Year,
    t => t.Year,
    (r, t) =>
      new
      {
        Year = r.Year,
        Champion = r.Name,
        Constructor = t.Name
      }).OrderBy(item => item.Year).Take(10);
  //...
}
```

结果显示了在同时有了赛车手冠军和车队冠军的前 10 年中，匿名类型中的数据：

```
Year World Champion Constructor Title
1958: Mike Hawthorn Vanwall
1959: Jack Brabham Cooper
1960: Jack Brabham Cooper
1961: Phil Hill Ferrari
1962: Graham Hill BRM
1963: Jim Clark Lotus
1964: John Surtees Ferrari
1965: Jim Clark Lotus
1966: Jack Brabham Brabham
1967: Denny Hulme Brabham
```

图 12-1 是通过内部连接结合的两个集合的图形表示。使用内部连接，结果与两个集合匹配。

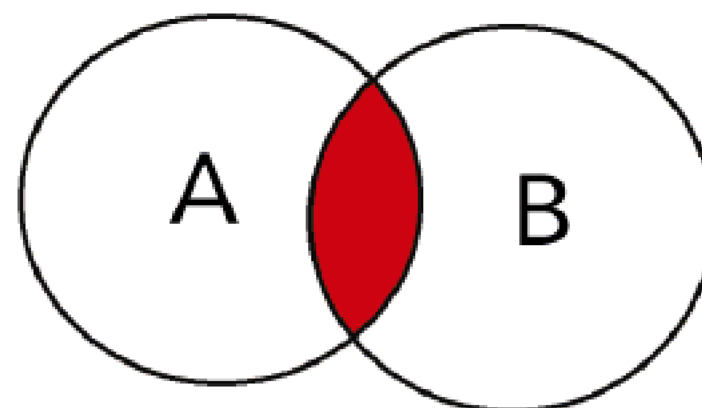



图 12-1

## 12.2.10 左外连接

上一个连接示例的输出从 1958 年开始，因为从这一年开始，才同时有了赛车手冠军和车队冠军。赛车手冠军出现得更早一些，是在 1950 年。使用内连接时，只有找到了匹配的记录才返回结果。为了在结果中包含所有的年份，可以使用左外连接。左外连接返回左边序列中的全部元素，即使它们在右边的序列中并没有匹配的元素。

下面修改前面的 LINQ 查询，使用左外连接。左外连接用 join 子句和 DefaultIfEmpty 方法定义。如果查询的左侧(赛车手)没有匹配的车队冠军，就使用 DefaultIfEmpty 方法定义其右侧的默认值(代码文件 EnumerableSample/JoinSamples.cs)：

```
static void LeftOuterJoin()
{
  //...
  var racersAndTeams =
    (from r in racers
```

```
        join t in teams on r.Year equals t.Year into rt
        from t in rt.DefaultIfEmpty()
        orderby r.Year
        select new
        {
          Year = r.Year,
          Champion = r.Name,
          Constructor = t == null ? "no constructor championship" : t.Name
        }).Take(10);
  //...
}
```

通过扩展方法执行相同的查询时，使用 GroupJoin 方法。前三个参数与 Join 和 GroupJoin 相似。但 GroupJoin 的结果是不同的。Join 方法返回一个平铺列表，而 GroupJoin 返回一个列表，其中第一个列表中包含的每个匹配项都包含第二个列表中的一个匹配列表。使用下面的 SelectMany 方法，列表再次被铺平。如果没有匹配的车队，则 Constructors 属性就赋予类型的默认值，对类二元，默认值都为空。创建匿名类型时，如果车队为空，Constructors 属性将赋予字符串“no constructor championship”(代码文件 EnumerableSample/JoinSamples.cs)：

```
static void LeftOuterJoinWithMethods()
{
  //...
  var racersAndTeams =
    racers.GroupJoin(
      teams,
      r => r.Year,
      t => t.Year,
      (r, ts) => new
      {
        Year = r.Year,
        Champion = r.Name,
        Constructors = ts
      })
      .SelectMany(
        rt => rt.Constructors.DefaultIfEmpty(),
        (r, t) => new
        {
          Year = r.Year,
          Champion = r.Champion,
          Constructor = t?.Name ?? "no constructor championship"
        });
  //...
}
```

**注意：**

GroupJoin 方法的其他用法详见下一节。

用这个查询运行应用程序，得到的输出将从 1950 年开始，如下所示：

```
Year Champion Constructor Title
1950: Nino Farina no constructor championship
1951: Juan Manuel Fangio no constructor championship
1952: Alberto Ascari no constructor championship
1953: Alberto Ascari no constructor championship
1954: Juan Manuel Fangio no constructor championship
1955: Juan Manuel Fangio no constructor championship
1956: Juan Manuel Fangio no constructor championship
1957: Juan Manuel Fangio no constructor championship
1958: Mike Hawthorn Vanwall
1959: Jack Brabham Cooper
```

图 12-2 是两个集合和一个左外连接的图形表示。使用左外连接，结果不仅与集合 A 和集合 B 匹配，还包括右集合 B。

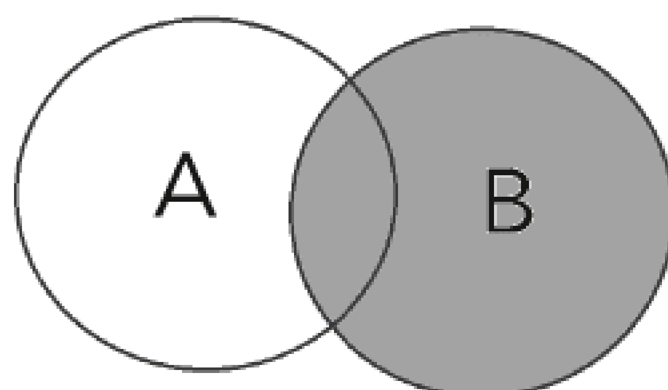



图 12-2

### 12.2.11 组连接

左外连接使用了组连接和into子句。它有一部分语法与组连接相同，只不过组连接不使用DefaultIfEmpty方法。

使用组连接时，可以连接两个独立的序列，对于其中一个序列中的某个元素，另一个序列中存在对应的一个项列表。

下面的示例使用了两个独立的序列。一个是前面例子中已经看过的冠军列表。另一个是一个 Championship 类型的集合。下面的代码段显示了 Championship 类型。该类包含冠军年份以及该年份中获得第一名、第二名和第三名的赛车手，对应的属性分别为 Year、First、Second 和 Third(代码文件 DataLib/Championship.cs)：

```
public class Championship
{
  public Championship(int year, string first, string second, string third)
  {
    Year = year;
    First = first;
    Second = second:
    Third = third;
  }

  public int Year { get; }
  public string First { get; }
  public string Second { get; }
  public string Third { get; }
}
```

GetChampionships 方法返回了冠军集合，如下面的代码段所示(代码文件 DataLib/Formula1.cs)：

```
private static List<Championship> s_championships;
public static IEnumerable<Championship> GetChampionships()
{
  if (s_championships == null)
  {
    s_championships = new List<Championship>
    {
      new Championship(1950, "Nino Farina", "Juan Manuel Fangio",
        "Luigi Fagioli"),
      new Championship(1951, "Juan Manuel Fangio", "Alberto Ascari",
        "Froilan Gonzalez"),
      //...
    }
  }
  return s_championships;
}
```

冠军列表应与每个冠军年份中获得前三名的赛车手构成的列表组合起来，然后显示每一年的结果。

因为冠军列表中的每一项都包含 3 个赛车手，所以首先需要把这个列表摊平。一种方式是使用复合的 from 子句。由于没有集合可用于单个项目的属性，而是需要将三个属性(First、Second 和 Third)合并、摊平，因此创建了一个新的 List<T>，其中填充了这些属性的信息。对于新建的对象，可以使用自定义类和匿名类型，如前所述。这次使用 C# 7 的一个新特性：创建一个元组。元组包含不同类型的成员，可以使用带括号的元组字面量创建，如下面的代码片段所示。这里，元组的一个摊平列表包含年份、冠军的位置、赛车手的名字和姓氏信息(代码文件 EnumerableSample/JoinSamples.cs)：

```
static void GroupJoin()
{
  var racers = from cs in Formula1.GetChampionships()
               from r in new List<
               (int Year, int Position, string FirstName, string LastName)>()
               {
               (cs.Year, Position: 1, FirstName: cs.First.FirstName(),
               LastName: cs.First.LastName()),
               (cs.Year, Position: 2, FirstName: cs.Second.FirstName(),
               LastName: cs.Second.LastName()),
               (cs.Year, Position: 3, FirstName: cs.Third.FirstName(),
               LastName: cs.Third.LastName())
               }
               select r;
  //...
}
```

**注意：**

第 13 章给出了元组的详细信息。这里的示例代码使用了 C# 7.1 中的元组增强，因此编译器设置必须配置为至少使用 7.1 版本。

扩展方法FirstName和LastName使用空格字符拆分字符串(代码文件EnumerableSample/StringExtensions.cs)：

```
public static class StringExtensions
{
  public static string FirstName(this string name) =>
    name.Substring(0, name.LastIndexOf(' '));

  public static string LastName(this string name) =>
    name.Substring(name.LastIndexOf(' ') + 1);
}
```

现在就可以连接两个序列。Formula1.GetChampions 返回一个 Racers 列表，racers 变量返回包含年份、比赛结果和赛车手姓名的一个元组。仅使用姓氏比较两个集合中的项是不够的。有时列表中可能同时包含了一个赛车手和他的父亲(如 Damon Hill 和 Graham Hill)，所以必须同时使用 FirstName 和 LastName 进行比较。这是通过为两个列表创建一个新的元组实现的。通过使用 into 子句，第二个集合中的结果被添加到变量 yearResults 中。对于第一个集合中的每一个赛车手，都创建了一个 yearResults，它包含了在第二个集合中匹配名和姓的结果。最后，用 LINQ 查询创建了一个包含所需信息的新元组类型(代码文件 EnumerableSample/JoinSamples.cs)：

```
static void GroupJoin()
{
  //...
  var q = (from r in Formula1.GetChampions()
           join r2 in racers on
           (
             r.FirstName,
             r.LastName
           )
           equals
           (
             r2.FirstName,
             r2.LastName
           )
           into yearResults
           select
           (
             r.FirstName,
             r.LastName,
             r.Wins,
             r.Starts,
             Results: yearResults
           ));

  foreach (var r in q)
  {
    Console.WriteLine($"{r.FirstName} {r.LastName}");
    foreach (var results in r.Results)
    {
      Console.WriteLine($"\t{results.Year} {results.Position}");
    }
  }
}
```

下面显示了 foreach 循环得到的最终结果。Jenson Button 3 次进入前三：2004 年是第三名，2009 年是第一名，2011 年是第一名。Sebastian Vettel 四次获得世界冠军，2009 年获得第二名，2015 年获得第三名。Nico Rosberg 获得 2016 年世界冠军，2014 年和 2015 年两次获得第二名：

```
Jenson Button
        2004 3
        2009 1
        2011 2
Sebastian Vettel
        2009 2
        2010 1
        2011 1
        2012 1
        2013 1
```

```
        2015 3
Nico Rosberg
        2014 2
        2015 2
        2016 1
```

使用 GroupJoin 和扩展方法，语法可能看起来更容易理解。首先，使用 SelectMany 方法完成复合的 from 子句。这一部分没有太大的不同，并且再次使用了元组。调用 GroupJoin 方法时，传递赛车手作为第一个参数，把冠军与摊平的赛车手连接起来，用第二个和第三个参数来匹配两个集合。第四个参数接收第一个集合和第二个集合的赛车手。结果包含位置和年份，被写入 Results 元组成员(代码文件 EnumerableSample/JoinSamples.cs)：

```
static void GroupJoinWithMethods()
{
  var racers = Formula1.GetChampionships()
    .SelectMany(cs => new List<(int Year, int Position, string FirstName,
       string LastName)>
    {
      (cs.Year, Position: 1, FirstName: cs.First.FirstName(),
        LastName: cs.First.LastName()),
      (cs.Year, Position: 2, FirstName: cs.Second.FirstName(),
        LastName: cs.Second.LastName()),
      (cs.Year, Position: 3, FirstName: cs.Third.FirstName(),
        LastName: cs.Third.LastName())
    });

  var q = Formula1.GetChampions()
    .GroupJoin(racers,
      r1 => (r1.FirstName, r1.LastName),
      r2 => (r2.FirstName, r2.LastName),
      (r1, r2s) => (r1.FirstName, r1.LastName, r1.Wins, r1.Starts,
        Results: r2s));
  //...
}
```

### 12.2.12　集合操作

扩展方法 Distinct()、Union()、Intersect()和 Except()都是集合操作。下面创建一个驾驶法拉利的一级方程式冠军序列和驾驶迈凯伦的一级方程式冠军序列，然后确定是否有驾驶法拉利和迈凯伦的冠军。当然，这里可以使用 Intersect()扩展方法。

首先获得所有驾驶法拉利的冠军。这只是一个简单的 LINQ 查询，其中使用复合的 from 子句访问 Cars 属性，该属性返回一个字符串对象序列。

```
var ferrariDrivers = from r in Formula1.GetChampions()
                     from c in r.Cars
                     where c == "Ferrari"
                     orderby r.LastName
                     select r;
```

现在建立另一个基本相同的查询，但 where 子句的参数不同，以获得所有驾驶迈凯伦的冠军。最好不要再次编写相同的查询。而可以创建一个方法，给它传递参数 car。如果在其他地方不需要该方法，就可以创建一个本地函数。racersByCar 是一个本地函数的名称，它实现为包含 LINQ 查询的 lambda 表达式。本地函数 racersByCar 在方法 SetOperations 的作用域内定义，因此只能在此方法中调用它。LINQ Intersect 扩展方法用于获取所有使用法拉利和迈凯伦赢得总冠军的赛车手(代码文件 EnumerableSample/LinqSamples.cs)：

```
static void SetOperations()
{
  IEnumerable<Racer> racersByCar(string car) =>
    from r in Formula1.GetChampions()
    from c in r.Cars
    where c == car
    orderby r.LastName
    select r;

  Console.WriteLine("World champion with Ferrari and McLaren");
  foreach (var racer in
    racersByCar("Ferrari").Intersect(racersByCar("McLaren")))
```

```
    {
      Console.WriteLine(racer);
    }
  }
```

**注意：**

本地函数是 C# 7 的一个新特性，详见第 13 章。

结果只有一个赛车手 Niki Lauda：

```
World champion with Ferrari and McLaren
Niki Lauda
```

**注意：**

集合操作通过调用实体类的 GetHashCode()和 Equals()方法来比较对象。对于自定义比较，还可以传递一个实现了 IEqualityComparer<T>接口的对象。在这里的示例中，GetChampions()方法总是返回相同的对象，因此默认的比较操作是有效的。如果不是这种情况，就可以重载集合方法来自定义比较操作。

### 12.2.13 合并

Zip()方法允许用一个谓词函数把两个相关的序列合并为一个。

首先，创建两个相关的序列，它们使用相同的筛选(国家意大利)和排序方法。对于合并，这很重要，因为第一个集合中的第一项会与第二个集合中的第一项合并，第一个集合中的第二项会与第二个集合中的第二项合并，以此类推。如果两个序列的项数不同，Zip()方法就在到达较小集合的末尾时停止。

第一个集合中的元素有一个 Name 属性，第二个集合中的元素有 LastName 和 Starts 两个属性。

在 racerNames 集合上使用 Zip()方法，需要把第二个集合(racerNamesAndStarts)作为第一个参数。第二个参数的类型是 Func<TFirst, TSecond, TResult>。这个参数实现为一个 lambda 表达式，它通过参数 first 接收第一个集合的元素，通过参数 second 接收第二个集合的元素。其实现代码创建并返回一个字符串，该字符串包含第一个集合中元素的 Name 属性和第二个集合中元素的 Starts 属性(代码文件 EnumerableSample/LinqSamples.cs)：

```
static void ZipOperation()
{
  var racerNames = from r in Formula1.GetChampions()
                   where r.Country == "Italy"
                   orderby r.Wins descending
                   select new
                   {
                     Name = r.FirstName + " " + r.LastName
                   };

  var racerNamesAndStarts = from r in Formula1.GetChampions()
                            where r.Country == "Italy"
                            orderby r.Wins descending
                            select new
                            {
                             r.LastName,
                             r.Starts
                            };

  var racers = racerNames.Zip(racerNamesAndStarts,
               (first, second) => first.Name + ", starts: second.Starts);

  foreach (var r in racers)
  {
    Console.WriteLine(r);
  }
}
```

这个合并的结果是：

```
Alberto Ascari, starts: 32
Nino Farina, starts: 33
```

### 12.2.14 分区

扩展方法 Take()和 Skip()等的分区操作可用于分页，例如，在第一个页面上只显示 5 个赛车手，在下一个页面上显示接下来的 5 个赛车手等。

在下面的 LINQ 查询中，把扩展方法 Skip()和 Take()添加到查询的最后。Skip()方法先忽略根据页面大小和实际页数计算出的项数，再使用 Take()方法根据页面大小提取一定数量的项(代码文件 EnumerableSample/LinqSamples.cs)：

```
static void Partitioning()
{
  int pageSize = 5;
  int numberPages = (int)Math.Ceiling(Formula1.GetChampions().Count() /
    (double)pageSize);

  for (int page = 0; page < numberPages; page++)
  {
    Console.WriteLine($"Page {page}");

    var racers = (from r in Formula1.GetChampions()
                  orderby r.LastName, r.FirstName
                  select r.FirstName + " " + r.LastName).
                 Skip(page * pageSize).Take(pageSize);

    foreach (var name in racers)
    {
      Console.WriteLine(name);
    }
    Console.WriteLine();
  }
}
```

下面输出了前 3 页：

```
Page 0
Fernando Alonso
Mario Andretti
Alberto Ascari
Jack Brabham
Jenson Button

Page 1
Jim Clark
Juan Manuel Fangio
Nino Farina
Emerson Fittipaldi
Mika Hakkinen

Page 2
Lewis Hamilton
Mike Hawthorn
Damon Hill
Graham Hill
Phil Hill
```

分页在 Windows 或 Web 应用程序中非常有用，可以只给用户显示一部分数据。

**注意：**

这个分页机制的一个要点是，因为查询会在每个页面上执行，所以改变底层的数据会影响结果。在继续执行分页操作时，会显示新对象。根据不同的情况，这对于应用程序可能有利。如果这个操作是不需要的，就可以只对原来的数据源分页，然后使用映射到原始数据上的缓存。

使用 TakeWhile()和 SkipWhile()扩展方法，还可以传递一个谓词，根据谓词的结果提取或跳过某些项。

### 12.2.15 聚合操作符

聚合操作符(如 Count、Sum、Min、Max、Average 和 Aggregate 操作符)不返回一个序列，而返回一个值。

Count()扩展方法返回集合中的项数。下面的 Count()方法应用于 Racer 的 Years 属性，来筛选赛车手，只返回获得冠军次数超过 3 次的赛车手。因为同一个查询中需要使用同一个计数超过一次，所以使用 let 子句定义了一个变量 numberYears(代码文件 EnumerableSample/LinqSamples.cs)：

```
static void AggregateCount()
{
  var query = from r in Formula1.GetChampions()
              let numberYears = r.Years.Count()
              where numberYears >= 3
              orderby numberYears descending, r.LastName
              select new
              {
                Name = r.FirstName + " " + r.LastName,
                TimesChampion = numberYears
              };

  foreach (var r in query)
  {
    Console.WriteLine($"{r.Name} {r.TimesChampion}");
  }
}
```

结果如下：

```
Michael Schumacher 7
Juan Manuel Fangio 5
Lewis Hamilton 4
Alain Prost 4
Sebastian Vettel 4
Jack Brabham 3
Niki Lauda 3
Nelson Piquet 3
Ayrton Senna 3
Jackie Stewart 3
```

Sum()方法汇总序列中的所有数字，返回这些数字的和。下面的 Sum()方法用于计算一个国家赢得比赛的总次数。首先根据国家对赛车手分组，再在新创建的匿名类型中，把 Wins 属性赋予某个国家赢得比赛的总次数(代码文件 EnumerableSample/LinqSamples.cs)：

```
static void AggregateSum()
{
  var countries = (from c in
                   from r in Formula1.GetChampions()
                   group r by r.Country into c
                   select new
                   {
                     Country = c.Key,
                     Wins = (from r1 in c
                             select r1.Wins).Sum()
                   }
                   orderby c.Wins descending, c.Country
                   select c).Take(5);

  foreach (var country in countries)
  {
    Console.WriteLine("{country.Country} {country.Wins}");
  }
}
```

根据获得一级方程式冠军的次数，最成功的国家是：

```
UK 216
Germany 162
Brazil 78
France 51
Finland 45
```

方法 Min()、Max()、Average()和 Aggregate()的使用方式与 Count()和 Sum()相同。Min()方法返回集合中的最小值，Max()方法返回集合中的最大值，Average()方法计算集合中的平均值。对于 Aggregate()方法，可以传递一个 lambda 表达式，该表达式对所有的值进行聚合。

### 12.2.16 转换操作符

本章前面提到，查询可以推迟到访问数据项时再执行。在迭代中使用查询时，查询会执行。而使用转换操作符会立即执行查询，把查询结果放在数组、列表或字典中。

在下面的例子中，调用 ToList()扩展方法，立即执行查询，得到的结果放在 List<T>类中(代码文件 EnumerableSample/LinqSamples.cs)：

```
static void ToList()
{
  List<Racer> racers = (from r in Formula1.GetChampions()
                        where r.Starts > 200
                        orderby r.Starts descending
                        select r).ToList();

  foreach (var racer in racers)
  {
    Console.WriteLine($"{racer} {racer:S}");
  }
}
```

查询结果显示，Jenson Button 是第一：

```
Jenson Button 306
Fernando Alonso 291
Michael Schumacher 287
Kimi Räikkönen 271
Nico Rosberg 207
Nelson Piquet 204
```

把返回的对象放在列表中并没有这么简单。例如，对于集合类中从赛车到赛车手的快速访问，可以使用新类 Lookup<TKey, TElement>。

**注意：**

Dictionary<TKey, TValue>类只支持一个键对应一个值。在 System.Linq 名称空间的类 Lookup<TKey, TElement>类中，一个键可以对应多个值。这些类详见第 10 章。

使用复合的 from 查询，可以摊平赛车手和赛车序列，创建带有 Car 和 Racer 属性的匿名类型。在返回的 Lookup 对象中，键的类型应是表示汽车的 string，值的类型应是 Racer。为了进行这个选择，可以给 ToLookup()方法的一个重载版本传递一个键和一个元素选择器。键选择器引用 Car 属性，元素选择器引用 Racer 属性(代码文件 EnumerableSample/LinqSamples.cs)：

```
static void ToLookup()
{
  var racers = (from r in Formula1.GetChampions()
                from c in r.Cars
                select new
                {
                  Car = c,
                  Racer = r
                }).ToLookup(cr => cr.Car, cr => cr.Racer);

  if (racers.Contains("Williams"))
  {
    foreach (var williamsRacer in racers["Williams"])
    {
      Console.WriteLine(williamsRacer);
    }
  }
}
```

用 Lookup 类的索引器访问的所有“Williams”冠军，结果如下：

```
Alan Jones
Keke Rosberg
Nigel Mansell
Alain Prost
Damon Hill
Jacques Villeneuve
```

如果需要在非类型化的集合上(如 ArrayList)使用 LINQ 查询，就可以使用 Cast()方法。在下面的例子中，基于 Object 类型的 ArrayList 集合用 Racer 对象填充。为定义强类型化的查询，可使用 Cast()方法(代码文件 EnumerableSample/LinqSamples.cs)：

```
static void ConvertWithCast
{
  var list = new System.Collections.ArrayList(Formula1.GetChampions()
    as System.Collections.ICollection);

  var query = from r in list.Cast<Racer>()
              where r.Country == "USA"
              orderby r.Wins descending
              select r;

  foreach (var racer in query)
  {
    Console.WriteLine("{racer:A}", racer);
  }
}
```

结果仅包含来自美国的一级方程式冠军：

```
Mario Andretti, country: USA, starts: 128, wins: 12
Phil Hill, country: USA, starts: 48, wins: 3
```

### 12.2.17　生成操作符

生成操作符 Range()、Empty()和 Repeat()不是扩展方法，而是返回序列的正常静态方法。在 LINQ to Objects 中，这些方法可用于 Enumerable 类。

有时需要填充一个范围的数字，此时就应使用 Range()方法。这个方法把第一个参数作为起始值，把第二个参数作为要填充的项数(代码文件 EnumerableSample/LinqSamples.cs)：

```
static void GenerateRange()
{
  var values = Enumerable.Range(1, 20);
  foreach (var item in values)
  {
    Console.Write($"{item} ", item);
  }
  Console.WriteLine();
}
```

当然，结果如下所示：

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

**注意：**

*Range()方法不返回填充了所定义值的集合，这个方法与其他方法一样，也推迟执行查询，并返回一个 RangeEnumerator，其中只有一条 yield return 语句，来递增值。*

可以把该结果与其他扩展方法合并起来，获得另一个结果，例如，使用 Select()扩展方法：

```
var values = Enumerable.Range(1, 20).Select(n => n * 3);
```

Empty()方法返回一个不返回值的迭代器，它可以用于需要一个集合的参数，其中可以给参数传递空集合。

Repeat()方法返回一个迭代器，该迭代器把同一个值重复特定的次数。

## 12.3　并行 LINQ

System.Linq 名称空间中包含的类 ParallelEnumerable 可以分解查询的工作，使其分布在多个线程上。尽管 Enumerable 类给 IEnumerable<T>接口定义了扩展方法，但 ParallelEnumerable 类的大多数扩展方法是 ParallelQuery<TSource>类的扩展。一个重要的例外是 AsParallel()方法，它扩展了 IEnumerable<TSource>接口，返回 ParallelQuery<TSource>类，所以正常的集合类可以以并行方式查询。

### 12.3.1 并行查询

为了说明 PLINQ (Parallel LINQ，并行 LINQ)，需要一个大型集合。对于可以放在 CPU 的缓存中的小集合，并行 LINQ 看不出效果。在下面的代码中，用随机值填充一个大型的 int 集合(代码文件 ParallelLinqSample/Program.cs)：

```
static IEnumerable<int> SampleData()
{
  const int arraySize = 50000000;
  var r = new Random();
  return Enumerable.Range(0, arraySize).Select(x => r.Next(140)).ToList();
}
```

现在可以使用 LINQ 查询筛选数据，进行一些计算，获取所筛选数据的平均数。该查询用 where 子句定义了一个筛选器，仅汇总对应值小于 20 的项，接着调用聚合函数 Sum()方法。与前面的 LINQ 查询的唯一区别是，这次调用了 AsParallel()方法。

```
static void LinqQuery(IEnumerable<int> data)
{
  var res = (from x in data.AsParallel()
             where Math.Log(x) < 4
             select x).Average();
  //...
}
```

与前面的 LINQ 查询一样，编译器会修改语法，以调用 AsParallel()、Where()、Select()和 Average()方法。AsParallel()方法用 ParallelEnumerable 类定义，以扩展 IEnumerable<T>接口，所以可以对简单的数组调用它。AsParallel()方法返回 ParallelQuery<TSource>。因为返回的类型，编译器选择的 Where()方法是 ParallelEnumerable.Where()，而不是 Enumerable.Where()。在下面的代码中，Select()和 Average()方法也来自 ParallelEnumerable 类。与 Enumerable 类的实现代码相反，对于 ParallelEnumerable 类，查询是分区的，以便多个线程可以同时处理该查询。集合可以分为多个部分，其中每个部分由不同的线程处理，以筛选其余项。完成分区的工作后，就需要合并，获得所有部分的总和。

```
static void ExtensionMethods(IEnumerable<int> data)
{
  var res = data.AsParallel()
    .Where(x => Math.Log(x) < 4)
    .Select(x => x).Average();
  //...
}
```

运行这行代码会启动任务管理器，这样就可以看出系统的所有 CPU 都在忙碌。如果删除 AsParallel()方法，就不可能使用多个 CPU。当然，如果系统上没有多个 CPU，就不会看到并行版本带来的改进。

### 12.3.2 分区器

AsParallel()方法不仅扩展了IEnumerable<T>接口，还扩展了Partitioner类。通过它，可以影响要创建的分区。

Partitioner 类用 System.Collection.Concurrent 名称空间定义，并且有不同的变体。Create()方法接受实现了 IList<T>类的数组或对象。根据这一点，以及 Boolean 类型的参数 loadBalance 和该方法的一些重载版本，会返回一个不同的 Partitioner 类型。对于数组，使用派生自抽象基类 OrderablePartitioner<TSource>的 DynamicPartitionerForArray<TSource>类和 StaticPartitionerForArray<TSource>类。

修改 12.3.1 小节中的代码，手工创建一个分区器，而不是使用默认的分区器(代码文件 ParallelLinqSample/Program.cs)：

```
static void UseAPartitioner(IList<int> data)
{
  var result = (from x in Partitioner.Create(data, true).AsParallel()
                where Math.Log(x) < 4
                select x).Average();
  //...
}
```

也可以调用 WithExecutionMode()和 WithDegreeOfParallelism()方法来影响并行机制。对于 WithExecutionMode()方法，可以传递 ParallelExecutionMode 的一个 Default 值或者 ForceParallelism 值。默认情况下，并行 LINQ 避免使用系统开销很高的并行机制。对于 WithDegreeOfParallelism()方法，可以传递一个整数值，以指定应并行运行的最大任务数。查询不应使用全部 CPU，这个方法会很有用。

**注意：**
任务和线程详见第 21 章。

### 12.3.3　取消

.NET 提供了一种标准方式，来取消长时间运行的任务，这也适用于并行 LINQ。

要取消长时间运行的查询，可以给查询添加 WithCancellation()方法，并传递一个 CancellationToken 令牌作为参数。CancellationToken 令牌从 CancellationTokenSource 类中创建。该查询在单独的线程中运行，在该线程中，捕获一个 OperationCanceledException 类型的异常。如果取消了查询，就触发这个异常。在主线程中，调用 CancellationTokenSource 类的 Cancel()方法可以取消任务(代码文件 ParallelLinqSample/Program.cs)。

```
static void UseCancellation(IEnumerable<int> data)
{
  var cts = new CancellationTokenSource();

  Task.Run(() =>
  {
    try
    {
      var res = (from x in data.AsParallel().WithCancellation(cts.Token)
                 where Math.Log(x) < 4
                 select x).Average();

      Console.WriteLine($"query finished, sum: {res}");
    }
    catch (OperationCanceledException ex)
    {
      Console.WriteLine(ex.Message);
    }
  });
  Console.WriteLine("query started");
  Console.Write("cancel? ");
  string input = ReadLine();
  if (input.ToLower().Equals("y"))
  {
    // cancel!
    cts.Cancel();
  }
}
```

**注意：**
关于取消和 CancellationToken 令牌的内容详见第 21 章。

## 12.4　表达式树

在 LINQ to Objects 中，扩展方法需要将一个委托类型作为参数，这样就可以将 lambda 表达式赋予参数。lambda 表达式也可以赋予 Expression<T>类型的参数。C#编译器根据类型给 lambda 表达式定义不同的行为。如果类型是 Expression<T>，编译器就从 lambda 表达式中创建一个表达式树，并存储在程序集中。这样，就可以在运行期间分析表达式树，并进行优化，以便于查询数据源。

下面看看前面使用的一个查询表达式：

```
var brazilRacers = from r in racers
                   where r.Country == "Brazil"
                   orderby r.Wins
                   select r;
```

这个查询表达式使用了扩展方法 Where()、OrderBy()和 Select()。Enumerable 类定义了 Where()扩展方法，并将委托类型 Func<T,bool>作为参数谓词。

```
public static IEnumerable<TSource> Where<TSource>(
  this IEnumerable<TSource> source, Func<TSource, bool> predicate);
```

这样，就把 lambda 表达式赋予谓词。这里 lambda 表达式类似于前面介绍的匿名方法。

```
Func<Racer, bool> predicate = r => r.Country == "Brazil";
```

Enumerable 类不是唯一一个定义了扩展方法 Where()的类。Queryable<T>类也定义了 Where()扩展方法。这个类对 Where()扩展方法的定义是不同的：

```
public static IQueryable<TSource> Where<TSource>(
  this IQueryable<TSource> source,
  Expression<Func<TSource, bool>> predicate);
```

其中，把 lambda 表达式赋予类型 Expression<T>，该类型的操作是不同的：

```
Expression<Func<Racer, bool>> predicate = r => r.Country == "Brazil";
```

除了使用委托之外，编译器还会把表达式树放在程序集中。表达式树可以在运行期间读取。表达式树从派生自抽象基类 Expression 的类中构建。Expression 类与 Expression<T>不同。继承自 Expression 类的表达式类有 BinaryExpression、ConstantExpression、InvocationExpression、lambdaExpression、NewExpression、NewArrayExpression、TernaryExpression 以及 Unary Expression 等。编译器会从 lambda 表达式中创建表达式树。

例如，lambda 表达式 r.Country == "Brazil"使用了 ParameterExpression、MemberExpression、ConstantExpression 和 MethodCallExpression，来创建一个表达式树，并将该树存储在程序集中。之后在运行期间使用这个树，创建一个用于底层数据源的优化查询。

在示例应用程序中，DisplayTree()方法在控制台上图形化地显示表达式树。其中传递了一个 Expression 对象，并根据表达式的类型，把表达式的一些信息写到控制台上。根据表达式的类型，递归地调用 DisplayTree()方法(代码文件 ExpressionTreeSample/Program.cs)。

```
static void DisplayTree(int indent, string message,
  Expression expression)
{
  string output = $"{string.Empty.PadLeft(indent, '>')} {message} " +
    $"! NodeType: {expression.NodeType}; Expr: {expression}";

  indent++;

  switch (expression.NodeType)
  {
    case ExpressionType.Lambda:
      Console.WriteLine(output);
      LambdaExpression lambdaExpr = (LambdaExpression)expression;
      foreach (var parameter in lambdaExpr.Parameters)
      {
        DisplayTree(indent, "Parameter", parameter);
      }
      DisplayTree(indent, "Body", lambdaExpr.Body);
      break;
    case ExpressionType.Constant:
      ConstantExpression constExpr = (ConstantExpression)expression;
      Console.WriteLine($"{output} Const Value: {constExpr.Value}");
      break;
    case ExpressionType.Parameter:
      ParameterExpression paramExpr = (ParameterExpression)expression;
      Console.WriteLine($"{output} Param Type: {paramExpr.Type.Name}");
      break;
    case ExpressionType.Equal:
    case ExpressionType.AndAlso:
    case ExpressionType.GreaterThan:
      BinaryExpression binExpr = (BinaryExpression)expression;
      if (binExpr.Method != null)
      {
        Console.WriteLine($"{output} Method: {binExpr.Method.Name}");
      }
      else
      {
        Console.WriteLine(output);
```

```
        }
        DisplayTree(indent, "Left", binExpr.Left);
        DisplayTree(indent, "Right", binExpr.Right);
        break;
      case ExpressionType.MemberAccess:
        MemberExpression memberExpr = (MemberExpression)expression;
        Console.WriteLine($"{output} Member Name: {memberExpr.Member.Name}, " +
          " Type: {memberExpr.Expression}");
        DisplayTree(indent, "Member Expr", memberExpr.Expression);
        break;
      default:
        Console.WriteLine();
        Console.WriteLine($"{expression.NodeType} {expression.Type.Name}");
        break;
    }
  }
```

**注意：**

*在方法 DisplayTree 中，没有处理所有的表达式类型，只处理了在下一个示例表达式中使用的类型。*

前面已经介绍了用于显示表达式树的表达式。这是一个 lambda 表达式，它有一个 Racer 参数，表达式体提取赢得比赛次数超过 6 次的巴西赛车手：

```
Expression<Func<Racer, bool>> expression =
  r => r.Country == "Brazil" && r.Wins > 6;
```

下面看看结果。lambda 表达式包含一个 Parameter 和一个 AndAlso 节点类型。AndAlso 节点类型的左边是一个 Equal 节点类型，右边是一个 GreaterThan 节点类型。Equal 节点类型的左边是 MemberAccess 节点类型，右边是 Constant 节点类型。

```
Lambda! NodeType: Lambda; Expr: r => ((r.Country == "Brazil") AndAlso (r.Wins > 6))
> Parameter! NodeType: Parameter; Expr: r Param Type: Racer
> Body! NodeType: AndAlso; Expr: ((r.Country == "Brazil") AndAlso (r.Wins > 6))
>> Left! NodeType: Equal; Expr: (r.Country == "Brazil") Method: op_Equality
>>> Left! NodeType: MemberAccess; Expr: r.Country Member Name: Country, Type: String
>>>> Member Expr! NodeType: Parameter; Expr: r Param Type: Racer
>>> Right! NodeType: Constant; Expr: "Brazil" Const Value: Brazil
>> Right! NodeType: GreaterThan; Expr: (r.Wins > 6)
>>> Left! NodeType: MemberAccess; Expr: r.Wins Member Name: Wins, Type: Int32
>>>> Member Expr! NodeType: Parameter; Expr: r Param Type: Racer
>>> Right! NodeType: Constant; Expr: 6 Const Value: 6
```

使用 Expression<T>类型的一个例子是 Entity Framework Core 和 WCF 数据服务的客户端提供程序。这些技术用 Expression<T>参数定义了扩展方法。这样，访问数据库的 LINQ 提供程序就可以读取表达式，创建一个运行期间优化的查询，从数据库中获取数据。

## 12.5 LINQ 提供程序

.NET 包含几个 LINQ 提供程序。LINQ 提供程序为特定的数据源实现了标准的查询操作符。LINQ 提供程序也许会实现比 LINQ 定义的更多的扩展方法，但至少要实现标准操作符。LINQ to XML 实现了一些专门用于 XML 的方法，例如，System.Xml.Linq 名称空间中的 Extensions 类定义的 Elements()、Descendants()和 Ancestors()方法。

LINQ 提供程序的实现方案是根据名称空间和第一个参数的类型来选择的。实现扩展方法的类的名称空间必须是开放的，否则扩展类就不在作用域内。在 LINQ to Objects 中定义的 Where()方法的参数和在 LINQ to Entities 中定义的 Where()方法的参数不同。

LINQ to Objects 中的 Where()方法用 Enumerable 类定义：

```
public static IEnumerable<TSource> Where<TSource>(
  this IEnumerable<TSource> source, Func<TSource, bool> predicate);
```

在 System.Linq 名称空间中，还有另一个类实现了操作符 Where。这个实现代码由 LINQ to Entities 使用。这些实现代码在 Queryable 类中可以找到：

```
public static IQueryable<TSource> Where<TSource>(
  this IQueryable<TSource> source,
  Expression<Func<TSource, bool>> predicate);
```

这两个类都在 System.Linq 名称空间的 System.Core 程序集中实现。那么，编译器如何选择使用哪个方法？表达式类型有什么用途？无论是用 Func<TSource, bool>参数传递，还是用 Expression <Func<TSource, bool>>参数传递，lambda 表达式都相同。只是编译器的行为不同，它根据 source 参数来选择。编译器根据其参数选择最匹配的方法。Entity Framework Core 的属性是 DbSet<TEntity>类型。DbSet<TEntity>实现了 IQueryable<TEntity>接口，因此 Entity Framework Core 使用 Queryable 类的 Where 方法。

## 12.6 小结

本章讨论了 LINQ 查询和查询所基于的语言结构，如扩展方法和 lambda 表达式，还列出了各种 LINQ 查询操作符，它们不仅用于筛选数据源，给数据源排序，还用于执行分区、分组、转换、连接等操作。

使用并行 LINQ 可以轻松地并行化运行时间较长的查询。

另一个重要的概念是表达式树。表达式树允许在运行期间构建对数据源的查询，因为表达式树存储在程序集中。表达式树的用法详见第 26 章。LINQ 是一个非常深奥的主题，更多的信息可查阅网上附加第 2 章。还可以下载其他第三方提供程序，例如，LINQ to MySQL、LINQ to Amazon、LINQ to Flickr、LINQ to LDAP 以及 LINQ to SharePoint。无论使用什么数据源，都可以通过 LINQ 使用相同的查询语法。

第 13 章将介绍函数式编程。许多较新的 C#特性都基于这种编程范式。

# 第 13 章

# C#函数式编程

**本章要点**

- 函数式编程概述
- 表达式体的成员
- 扩展方法
- using static 声明
- 本地函数
- 元组
- 模式匹配

**本章源代码下载：**

打开 www.wrox.com 的 Download Code 选项卡可下载本章源代码。源代码也可以在 HelloWorld 目录的 https://github.com/ProfessionalCSharp/ProfessionalCSharp7 中找到。

本章代码分为以下几个主要的示例文件：

- ExpressionBodiedMembers
- LocalFunctions
- Tuples
- PatternMatching

## 13.1　概述

C#从来都不是纯面向对象的编程语言。从一开始，C#就是面向组件的编程语言。面向组件是什么意思？C#提供了面向对象编程语言也在使用的继承和多态性；此外，它还通过特性提供对属性、事件和注释的本机支持。随后带有 LINQ 和表达式的版本也包括了声明性编程。使用声明式 LINQ 表达式，编译器会保存一个表达式树，该表达式树稍后由提供程序用于动态生成 SQL 语句。

**注意：**

C#的面向对象特性在第 4 章中讨论过，第 8 章包含了事件，LINQ 在第 12 章中介绍。

C#并不仅仅是单一的编程语言范例。相反，目前使用 C#创建应用程序的实用功能已添加到 C#的语法中。在过去的几年里，还添加了与函数式编程相关的更多特性。

函数式编程的基础是什么？函数式编程的最重要概念基于两种方法：避免状态突变和将函数作为一流的概念。接下来的两节将详细介绍这两种方法。

**注意：**
本章并没有给出用纯函数编程范式来编写应用程序的所有信息。这需要一整本书的篇幅。(如果想用这个范例编写程序，就应该考虑使用 F#编程语言，而不是使用 C#。)本章将采用编程的方式——与 C#一样。函数式编程使用的一些特性对所有应用程序类型都很有用；这就是 C#中提供这些特性的原因。随着时间的推移，越来越多的函数式编程功能将会以符合 C#编程风格的方式添加到 C#中。

### 13.1.1 避免状态突变

编程语言 F#是一种函数优先的语言，使用它创建自定义类型时，这种类型的对象默认是不可变的。对象可以在构造函数中初始化，但以后不能修改。如果需要可变性，该类型就需要显式地声明为可变的。这与 C#不同。

在 C#中，一些预定义类型是不可变的，比如 string 类型。用于更改字符串的方法总是返回一个新字符串。集合是不可变的吗？LINQ 使用的方法不会更改集合。相反，像 Where 和 OrderBy 这样的方法会返回一个已过滤的新集合，以及一个排序的新集合。

另一方面，List<T>集合提供了以可变方式实现的排序方法；原始集合是排好序的。为了得到更大的不变性，.NET 在名称空间 System.Collections.Immutable 中提供了完全不可变的集合。这些集合不提供更改集合的方法。相反，总是返回新的集合。

使用不可变类型的优点是什么？因为它保证没有人可以更改实例，所以可以使用多个线程并发地访问它，而不需要同步。对于不可变的类型，创建单元测试也更容易。

为了创建自定义类型，在 C# 6 中添加了一些特性，以创建不可变的类型。自从 C# 6 开始，就能够创建只带 get、自动实现的只读属性：

```
public string FirstName { get; }
```

这样，编译器就会创建一个只读字段和一个属性，该字段只能在构造函数中初始化，该属性可以使用 get 访问器返回该字段。

**注意：**
字符串在第 9 章中介绍。不可变集合在第 11 章中介绍。

由于某些库的需求，可以使用不可变类型的地方是有限的。在过去的几年里，越来越多的地方已经移除了限制。例如，NuGet 包 Newtonsoft.Json 允许使用不可变类型进行 JSON 序列化和反序列化。这个库使用构造函数来匹配创建实例所需的参数。Entity Framework 在过去几年里就是这样一个限制。然而，自从 Entity Framework Core 1.1 以来，表列可以映射到字段上，而不是读/写属性。

**注意：**
JSON 序列化包含在网上附加第 2 章中。Entity Framework Core 在第 26 章介绍。线程和同步在第 21 章中讨论。

**注意：**
本章不介绍创建不可变类型的 C#特性，因为这已经在第 3 章中讨论过了。C#允许使用 get 访问器创建自动实现的属性，编译器在其中创建了一个只读字段和一个返回该字段值的 get 访问器。C#的未来版本计划有更多的特性来创建不可变的类型，比如 records。

### 13.1.2　函数作为第一个类

使用函数式编程，函数是第一个类。这意味着函数可以用作函数的参数，函数可以从函数中返回，函数可以赋给变量。

这在 C#中一直是可能的：委托可以保留函数的地址，委托可以用作方法的参数，并且可以从方法中返回委托。但需要注意，将正常函数的调用与委托的调用进行比较，委托有一些相关的开销。有了委托，就会创建一个委托类的实例，这个实例包含方法引用的集合。调用委托时，会迭代集合，来调用分配给委托的每个方法。

> **注意：**
> 委托在第 8 章中介绍。

#### 1. 高阶函数

函数式编程定义了高阶函数，这种函数将另一个函数作为参数，或者返回一个函数。一些函数既将另一个函数作为参数，又返回一个函数。在 C#实现中，委托用作方法的参数和返回类型。

高阶函数的例子是为 LINQ 定义的方法，如上一章所述。例如，Where 方法接收 Func<TSource, bool>谓词：

```
public static IEnumerable<TSource> Where(this IEnumerable<TSource> source,
  Func<TSource, bool> predicate);
```

高阶函数既可以接收函数作为参数，也可以返回一个函数，参见本章后面的内容。

#### 2. 纯函数

函数式编程定义了术语“纯函数”。如果可能，应该优先考虑纯函数。纯函数满足两个要求：

- 纯函数始终给传递的相同参数返回相同的结果。
- 纯函数不产生副作用，如改变状态，或依赖外部资源。

当然，并不是所有的方法都可以实现为纯函数。纯函数的优点是测试很容易；没有外部依赖。

在创建访问外部源的方法时，可能会考虑将该方法分为两部分：一部分是纯函数，可能有复杂的逻辑，一部分不是纯函数。

了解了函数式编程的重要概念后，就该讨论 C#如何帮助理解这些概念的语法细节。

## 13.2　表达式体的成员

C# 6 允许表达式体成员的方法和属性只定义 get 访问器，而在 C# 7 中，只要在实现代码中只使用一条语句，表达式体成员就可以在任何地方使用。在函数式编程中，许多方法都只是一行代码，因此可以经常使用这个特性；代码行数减少，是因为不需要花括号。

> **注意：**
> 本书的其他章节已经介绍了这一特性，如第 3 章中的表达式体属性和表达式体方法，第 8 章中表达式体的事件访问器，因此本章没有涉及它们的每个方面。

看看下面的代码片段，其中，表达式体成员与属性访问器(get 和 set)一起使用，并使用 ToString 方法的实现，以及构造函数的实现。构造函数定义为接受 name 作为字符串参数，并要求将该字符串拆分为姓和名。这是用一条语句完成的，首先将字符串分割成一个字符串数组，然后使用这个字符串数组通过 out 参数来提取两个字符串_firstName 和_lastName (代码文件 ExpressionBodiedMembers/Person.cs)：

```
public class Person
{
  public Person(string name) =>
    name.Split(' ').ToStrings(out _firstName, out _lastName);

  private string _firstName;
```

```
  public string FirstName
  {
    get => _firstName;
    set => _firstName = value;
  }

  private string _lastName;
  public string LastName
  {
    get => _lastName;
    set => _lastName = value;
  }

  public override string ToString() => $"{FirstName} {LastName}";
}
```

在下面的代码片段中，自定义 out 参数由扩展方法 ToStrings 填充。这是字符串数组的扩展方法，它将数组元素移动到输出参数中(代码文件 ExpressionBodiedMembers/StringArrayExtensions.cs)：

```
public static class StringArrayExtensions
{
  public static void ToStrings(this string[] values, out string value1,
    out string value2)
  {
    if (values == null) throw new ArgumentNullException(nameof(values));
    if (values.Length != 2) throw new IndexOutOfRangeException(
      "only arrays with 2 values allowed");

    value1 = values[0];
    value2 = values[1];
  }
}
```

有了这些，就可以用包含一个字符串的姓名来创建 Person，通过 FirstName 和 LastName 属性访问该姓名(代码文件 ExpressionBodiedMembers/Program.cs)：

```
Person p = new Person("Katharina Nagel");
Console.WriteLine($"{p.FirstName} {p.LastName}");
```

# 13.3 扩展方法

扩展方法已经在第 12 章中讨论过了，本章的前一节实现了一个自定义扩展方法。但是，由于扩展方法对函数式编程概念有很大帮助，因此这里展示另一个例子。

在函数式编程中，许多方法都非常短，只包含单个语句，而前面所示的表达式体成员有助于减少代码行数。例如，可以将 using 语句改为方法。下面的扩展方法名为 Use，它是所有实现 IDisposable 接口的类的扩展方法。using 语句在实现中使用，以在使用后释放该项。对于该项的用户，可以将 Action<T>委托传递给 Use 方法(代码文件 UsingStatic/FunctionalExtensions.cs)。

```
public static class FunctionalExtensions
{
  public static void Use<T>(this T item, Action<T> action)
    where T : IDisposable
  {
    using (item)
    {
      action(item);
    }
  }
}
```

实现接口 IDisposable 的示例类是使用 Resource 类定义的。这个类提供了 Foo 方法和 IDisposable 功能(代码文件 UsingStatic/Resource.cs)：

```
class Resource : IDisposable
{
  public void Foo() => Console.WriteLine("Foo");

  private bool disposedValue = false;
```

```
  protected virtual void Dispose(bool disposing)
  {
    if (!disposedValue)
    {
      if (disposing)
      {
        Console.WriteLine("release resource");
      }
      disposedValue = true;
    }
  }

  public void Dispose() => Dispose(true);
}
```

现在看看访问这个 Resource 对象的经典 using 语句块：

```
using (var r = new Resource())
{
  r.Foo();
}
```

在 Use 方法中，可以在单个语句中访问和释放资源(代码文件 UsingStatic/Program.cs)：

```
new Resource().Use(r => r.Foo());
```

## 13.4　using static 声明

许多实际的扩展可以通过扩展方法来实现，比如前面的 Use 扩展方法或第 12 章介绍的用于 LINQ 的许多扩展方法。在本书后面的许多章节中还将介绍.NET 提供的许多扩展方法。

并非所有实际的扩展都有可以扩展的类型。对于某些场景，简单的静态方法比较适合。为了更容易调用这些方法，可以使用 using static 声明除去类名。

例如，如果打开了 System.Console

```
using static System.Console;
```

可以把下面的代码

```
Console.WriteLine("Hello  World!");
```

改为

```
WriteLine("Hello World!");
```

在使用此声明之后，就可以使用类 Console 的所有静态成员，如 WriteLine、Write、ReadLine、Read、Beep 等，而不需要编写 Console 类。只需要确保在打开其他类的静态成员时不要陷入冲突，或者在使用静态方法时不要使用基类的方法。

下面看一个实际的例子。高阶函数以函数作为参数，或者返回一个函数，或者返回两个函数。在处理函数时，可以将两个函数合并到一个函数中。

为此可以使用 Compose 方法，如下面的代码片段所示(代码文件 UsingStatic/FunctionalExtensions.cs)：

```
public static class FunctionalExtensions
{
  //...
  public static Func<T1, TResult> Compose<T1, T2, TResult>(
    Func<T1, T2> f1, Func<T2, TResult> f2) =>
      a => f2(f1(a));
}
```

该泛型方法定义了三个类型参数和两个委托类型 Func 的参数。请记住，委托 Func<T, TResult>引用了一个带有单个参数的方法，其返回类型可以是不同的类型。Compose 方法接受两个 Func 参数，把两个方法组合到一个方法中。传递给 Compose 的第一个方法(f1)可以有两个不同的类型，一个用于输入 T，另一个用于输出(T2)，而传递的第二个方法(f2)所需要的输入类型(T2)与第一个方法的输出类型(T2)相同，并且可以有不同的输出类型(TResult)。Compose 方法本身返回一个 Func 委托，其输入类型与第一个方法相同(T)，输出类型与第二个方法相同(TResult)。实现可能看起来有点可怕，因为后面跟着连续两个 lambda 操作符。理解了方法返回的内容（一

个方法)时，这个构造就将变得清晰。返回的方法是 Func<T1, TResult>。在第一个 lambda 操作符之后，=> f2(f1(a));定义了这个方法。变量的类型为 T1，返回的方法类型为 TResult，与 f2 返回的结果类型相同，f2 以输入作为参数接收 f1。

要使用 Compose 方法，首先创建两个委托 f1 和 f2，在输入中添加 1 或 2。这些委托会与 Compose 方法相结合。由于 usingstatic 声明打开了类 FunctionalExtensions 的静态成员，所以可以不使用类名来调用 Compose 方法。在使用 Compose 方法创建 f3 之后，就调用 f3 方法(代码文件 UsingStatic/Program.cs)：

```
using System;
using static System.Console;
using static UsingStatic.FunctionalExtensions;

namespace UsingStatic
{
  class Program
  {
    static void Main()
    {
      //...

      Func<int, int> f1 = x => x + 1;
      Func<int, int> f2 = x => x + 2;
      Func<int, int> f3 = Compose(f1, f2);
      var x1 = f3(39);
      WriteLine(x1);
      //...
    }
  }
}
```

写入控制台的结果当然是 42。

声明 Compose 方法时，参数类型可以在输入和输出之间有所不同。在下面的代码片段中，传递给 Compose 方法的第一个方法接收一个字符串，并返回 Person 对象；第二个方法接收 Person 并返回一个字符串。如果编译器不能从变量和返回类型中识别参数类型，就必须指定具体的委托类型，方法是接收字符串并返回一个 Person。只有变量名，并不能帮助编译器确定它的类型。通过传递给 Compose 方法的第二个方法，显然，输入的类型与第一个方法返回的类型相同，因此不需要指定类型。在调用 Compose 方法之后，变量 greetPerson 是两个输入方法的组合：

```
var greetPerson = Compose(
  new Func<string, Person>(name => new Person(name)),
  person => $"Hello, {person.FirstName}");

WriteLine(greetPerson("Mario Andretti"));
```

在 WriteLine 方法中使用字符串 Mario Andretti 调用 greetPerson 方法，将字符串 Hello, Mario 写入控制台。

# 13.5 本地函数

C# 7 的一个新特性是本地函数：方法可以在方法中声明。本地函数在方法的作用域、属性访问器、构造函数或者 lambda 表达式内声明。本地函数只能在包含成员的作用域内调用。可以使用本地函数，而不是使用仅一个地方需要的私有方法。

下面是一个示例，且在没有本地函数的情况下启动——lambda 表达式将在下一个回合中由本地函数替换。下面的代码片段声明了分配给委托变量 add 的 lambda 表达式。变量 add 是在方法 IntroWithLambdaExpression 的作用域内，因此它只能在这个方法中调用(代码文件 LocalFunctions/Program.cs)：

```
private static void IntroWithLambdaExpression()
{
  Func<int, int, int> add = (x, y) =>
  {
    return x + y;
  };
  int result = add(37, 5);
  Console.WriteLine(result);
}
```

可以定义本地函数，而不是声明 lambda 表达式。本地函数的声明方式与普通方法类似，都带有返回类型、名称和参数。本地函数的调用方式与前面显示的 lambda 表达式相同：

```
private static void IntroWithLocalFunctions()
{
  int add(int x, int y)
  {
    return x + y;
  }
  int result = add(37, 5);
  Console.WriteLine(result);
}
```

与 lambda 表达式相比，本地函数的语法更简单，执行得也更出色。委托需要一个类的实例和一个引用的集合，而本地函数只需要对函数的一个引用，这个函数可以直接调用。开销和其他方法一样。

当然，如果本地函数可以通过单个语句来实现，则可以使用一个表达式体成员实现该功能：

```
private static void IntroWithLocalFunctionsWithExpressionBodies()
{
  int add(int x, int y) => x + y;

  int result = add(37, 5);
  Console.WriteLine(result);
}
```

在方法体中，本地函数可以在任何位置实现。没有必要将它们在方法体的顶部实现；也可以在其他地方实现，在此之前可以调用本地函数。这种行为与普通的方法一样。但是，与普通方法不同，本地函数不能是 virtual、abstract、private，也不能使用其他修饰符。唯一允许使用的修饰符是 async 和 unsafe。

与 lambda 表达式一样，本地函数可以从外部作用域(也称为闭包)中访问变量，如下面的代码片段所示，其中本地函数访问变量 z，这是在本地函数的外部定义的：

```
private static void IntroWithLocalFunctionsWithClosures()
{
  int z = 3;
  int result = add(37, 5);
  Console.WriteLine(result);

  int add(int x, int y) => x + y + z;
}
```

**注意：**
本地函数允许使用的唯一修饰符是 async 和 unsafe。第 15 章解释了 async 修饰符，第 17 章解释了 unsafe 修饰符。

使用本地函数的一个原因是，只需要在方法(或属性、构造函数等) 的作用域内使用功能。使用本地函数还有其他选择。性能是使用本地函数而不是 lambda 表达式的一个好理由。将本地函数与普通私有方法进行比较，本地函数没有性能优势。当然，本地函数可以使用闭包，而私有方法不能。这是使用本地函数的充分理由吗？要了解本地函数的真正优势，需要看到一些有用的示例，如下一节所述。

## 13.5.1　本地函数与 yield 语句

第 12 章包含了 Where 方法的一个简单实现，其中使用了 yield 语句。没有讨论的是参数的检查。下面将其添加到 Where1 方法的实现中，检查 source 和 predicate 参数是否为 null(代码文件 LocalFunctions/EnumerableExtensions.cs)：

```
public static IEnumerable<T> Where1<T>(this IEnumerable<T> source,
  Func<T, bool> predicate)
{
  if (source == null) throw new ArgumentNullException(nameof(source));
  if (predicate == null) throw new ArgumentNullException(nameof(predicate));

  foreach (T item in source)
  {
    if (predicate(item))
    {
      yield return item;
```

```
      }
    }
  }
```

编写代码测试 ArgumentNullException，定义预处理语句#line，以从源代码行 1000 开始。在第 1004 行中没有发生异常，其中 null 传递给 Where 1 方法；相反，在第 1006 行中包含的 foreach 语句发生了异常。延迟发现这个错误的原因是，在方法 Where 1 的实现中，延迟执行了 yield 语句(代码文件 LocalFunctions /Program.cs)：

```
private static void YieldSampleSimple()
{
#line 1000
  Console.WriteLine(nameof(YieldSampleSimple));
  try
  {
    string[] names = { "James", "Niki", "John", "Gerhard", "Jack" };
    var q = names.Where1(null);

    foreach (var n in q)  // callstack position for exception
    {
      Console.WriteLine(n);
    }
  }
  catch (ArgumentNullException ex)
  {
    Console.WriteLine(ex);
  }
  Console.WriteLine();
}
```

为了解决这个问题，并向调用者更早地提供错误信息，Where1 方法通过 Where2 方法在两个部分实现。这里，Where2 方法只检查不正确的参数，不包括 yield 语句。使用 yield return 的实现是在一个单独的私有方法 WhereImpl 中完成的。在检查输入参数之后，从 Where2 方法中调用此方法(代码文件 LocalFunctions/EnumerationExtensions.cs)。

```
public static IEnumerable<T> Where2<T>(this IEnumerable<T> source,
  Func<T, bool> predicate)
{
  if (source == null) throw new ArgumentNullException(nameof(source));
  if (predicate == null) throw new ArgumentNullException(nameof(predicate));

  return Where2Impl(source, predicate);
}

private static IEnumerable<T> Where2Impl<T>(IEnumerable<T> source,
  Func<T, bool> predicate)
{
  foreach (T item in source)
  {
    if (predicate(item))
    {
      yield return item;
    }
  }
}
```

现在调用该方法，堆栈跟踪显示在第 1004 行中发生的错误，其中调用了 Where 2 方法(代码文件 LocalFunctions/Program.cs)：

```
private static void YieldSampleWithPrivateMethod()
{
#line 1000
  Console.WriteLine(nameof(YieldSampleWithPrivateMethod));
  try
  {
    string[] names = { "James", "Niki", "John", "Gerhard", "Jack" };
    var q = names.Where2(null);  // callstack position for exception

    foreach (var n in q)
    {
      Console.WriteLine(n);
    }
  }
  catch (ArgumentNullException ex)
  {
```

```
    Console.WriteLine(ex);
  }
  Console.WriteLine();
}
```

这个问题是用 Where2 方法解决的。但是，现在有了一个仅需要在一个地方使用的私有方法。Where2 方法的主体包括参数检查和 Where2Impl 方法的调用。对于私有方法来说，这是一个很好的场景。Where3 方法的实现包括对输入参数的检查（与以前一样），以及一个私有函数，而不是以前的私有方法 Where2Impl。本地函数可以有更简单的签名，因为它可以从外部作用域访问变量的源和谓词(代码文件 LocalFunctions/EnumerableExtensions.cs):

```
public static IEnumerable<T> Where3<T>(this IEnumerable<T> source,
  Func<T, bool> predicate)
{
  if (source == null) throw new ArgumentNullException(nameof(source));
  if (predicate == null) throw new ArgumentNullException(nameof(predicate));

  return Iterator();

  IEnumerable<T> Iterator()
  {
    foreach (T item in source)
    {
      if (predicate(item))
      {
        yield return item;
      }
    }
  }
}
```

调用 Where3 方法，其结果与调用 Where2 方法的结果相同。堆栈跟踪显示了调用 Where3 方法的问题。

### 13.5.2　递归本地函数

另一个使用本地函数的场景是递归调用，如下面使用 QuickSort 方法的示例所示。这里，本地函数 Sort 是递归调用的，直到集合排好序为止(代码文件 LocalFunctions/Algorithms.cs):

```
public static void QuickSort<T>(T[] elements) where T : IComparable<T>
{
  void Sort(int start, int end)
  {
    int i = start, j = end;
    var pivot = elements[(start + end) / 2];

    while (i <= j)
    {
      while (elements[i].CompareTo(pivot) < 0) i++;
      while (elements[j].CompareTo(pivot) > 0) j--;
      if (i <= j)
      {
        T tmp = elements[i];
        elements[i] = elements[j];
        elements[j] = tmp;
        i++;
        j--;
      }
    }
    if (start < j) Sort(start, j);
    if (i < end) Sort(i, end);
  }

  Sort(0, elements.Length - 1);
}
```

**注意：**

在使用 C#时，需要小心使用递归调用。下面的递归循环在大约 24 000 次迭代之后，由于堆栈溢出而结束。C#编译器不像 F#编译器那样实现尾调用优化。而使用尾调用优化，递归调用会转换为迭代，以不消耗这个堆栈空间。

```
public static void WhenDoesItEnd()
{
  Console.WriteLine(nameof(WhenDoesItEnd));
  void InnerLoop(int ix)
  {
    Console.WriteLine(ix++);
    InnerLoop(ix);
  }
  InnerLoop(1);
}
```

# 13.6 元组

元组能够组合不同类型的对象。使用数组可以组合相同类型的对象，而元组允许使用类型的不同组合。元组有助于减少以下两个需求：

- 定义自定义类或结构，以返回多个值
- 定义参数，从方法中返回多个值

自从.NET Framework 4.0 版本以来，元组就以泛型 Tuple 类的形式存在。然而，它们并没有得到广泛使用，因为元组的不同对象可以通过 Item1、Item2、Item3 等属性访问，这既不吸引人，也没有提供任何关于其含义的信息。

这在 C# 7 中发生了变化，C# 7 提供了在编程语言中集成的元组功能，这有了很大的改进，如下一个示例所示，它使用了一个简单的不可变的 Person 类(代码文件 TuplesSample/Person.cs)：

```
public class Person
{
  public Person(string firstName, string lastName)
  {
    FirstName = firstName;
    LastName = lastName;
  }
  public string FirstName { get; }
  public string LastName { get; }

  public override string ToString() => $"{FirstName} {LastName}";
  //...
}
```

## 13.6.1 元组的声明和初始化

可以使用圆括号声明一个元组，并使用通过括号创建的元组字面量来初始化。在下面的代码片段中，左侧声明了一个元组变量 t，其中包含一个字符串、一个 int 和一个 Person。右边使用一个元组字面量来创建一个元组，它包含字符串 magic、数字 4，以及使用 Person 类的构造函数初始化的 Person 对象。访问元组时，可以使用变量 t 以及在括号中声明的成员(本例中为 s、i 和 p)。(代码文件 Tuples/Program.cs)：

```
private static void IntroTuples()
{
  (string s, int i, Person p) t = ("magic", 42, new Person(
    "Stephanie", "Nagel"));
  Console.WriteLine($"s: {t.s}, i: {t.i}, p: {t.p}");
  //...
}
```

运行应用程序时，输出显示了元组的值：

```
s: magic, i: 42, p: Stephanie Nagel
```

元组字面量也可以分配给元组变量，而不需要声明它的成员。这样，元组的成员就可以使用 ValueTuple 结构的成员名称来访问：Item1、Item2 和 Item3。

```
private static void IntroTuples()
{
  //...
  var t2 = ("magic", 42, new Person("Matthias", "Nagel"));
  Console.WriteLine($"string: {t2.Item1}, int: {t2.Item2},
    person: {t2.Item3}");
```

```
  //...
}
```

可以通过定义名称后跟冒号，来为元组字面量中的元组指定名称，它与对象字面量的语法相同：

```
private static void IntroTuples()
{
  //...
  var t3 = (s: "magic", i: 42, p: new Person("Matthias", "Nagel"));
  Console.WriteLine($"s: {t3.s}, i: {t3.i}, p: {t3.p}");
  //...
}
```

有了这些，名字只是一种方便的方式。当类型匹配时，可以将一个元组分配给另一个元组；名字并不重要：

```
private static void IntroTuples()
{
  //...
  (string astring, int anumber, Person aperson) t4 = t3;
  Console.WriteLine($"s: {t4.astring}, i: {t4.anumber}, p: {t4.aperson}");
}
```

### 13.6.2 元组解构

还可以将元组分解为变量。为此，只需要从前面的代码示例中删除元组变量，并在括号中定义变量名。然后可以直接访问变量，其中包含元组部分的值(代码文件 Tuples/Program.cs)：

```
private static void TupleDeconstruction()
{
  (string s, int i, Person p) = ("magic", 42, new Person("Stephanie",
    "Nagel"));
  Console.WriteLine($"s: {s}, i: {i}, p: {p}");
  //...
}
```

还可以使用 var 关键字声明解构的变量；类型由元组字面量定义。还可以在初始化之前声明变量，并将元组分解为现有变量：

```
private static void TupleDeconstruction()
{
  //...
  (var s1, var i1, var p1) = ("magic", 42, new Person("Stephanie", "Nagel"));
  Console.WriteLine($"s: {s1}, i: {i1}, p: {p1}");

  string s2;
  int i2;
  Person p2;
  (s2, i2, p2) = ("magic", 42, new Person("Katharina", "Nagel"));
  Console.WriteLine($"s: {s2}, i: {i2}, p: {p2}");
  //...
}
```

如果不需要元组的所有部分，可以使用_忽略该部分，如下所示：

```
private static void TupleDeconstruction()
{
  //...
  (string s3, _, _) = ("magic", 42, new Person("Katharina", "Nagel"));
  Console.WriteLine(s3);
}
```

**注意：**

在不需要结果的情况下，使用 out 参数修饰符调用方法时，可能已经使用了_。在这个场景中，使用_只是一个命名约定。给元组使用_是不同的。不需要声明类型，_可以使用多次；它是一个编译器特性，解构时可以忽略这部分。

### 13.6.3 元组的返回

下面介绍一个更有用的示例：返回元组的方法。在下面的代码片段中，Divide 方法接收两个参数，并返回一个由两个 int 值组成的元组。结果用一个元组字面量返回(代码文件 Tuples/Program.cs)：

```
static (int result, int remainder) Divide(int dividend, int divisor)
{
  int result = dividend / divisor;
  int remainder = dividend % divisor;
  return (result, remainder);
}
```

结果分解为 result 和 remainder 变量：

```
private static void ReturningTuples()
{
  (int result, int remainder) = Divide(7, 2);
  Console.WriteLine($"7 / 2 - result: {result}, remainder: {remainder}");
}
```

**注意：**

使用元组，可以避免通过 out 参数声明方法签名。out 参数不能与 async 方法一起使用；此限制不适用于元组。

### 13.6.4 幕后的原理

使用新的元组语法，C#编译器在后台创建 ValueTuple 结构，.NET 为 1 到 7 个泛型参数定义了多个 ValueTuple 结构，还定义了另一个 ValueTuple 结构，其中第 8 个参数可以是另一个元组。使用元组字面量会调用 Tuple.Create。Tuple 结构定义了名为 Item1、Item2、Item3 等的字段，以访问所有项(代码文件 Tuples/Program.cs)：

```
private static void BehindTheScenes()
{
  (string s, int i) t1 = ("magic", 42); // tuple literal
  Console.WriteLine($"{t1.s} {t1.i}");
  ValueTuple<string, int> t2 = ValueTuple.Create("magic", 42);
  Console.WriteLine($"{t2.Item1}, {t2.Item2}");
}
```

字段的命名是如何从方法返回元组的过程中确定的？Divide 方法签名如下所示，

```
public static (int result, int remainder) Divide(int dividend, int divisor)
```

该方法签名转换为：返回 ValueTuple，TupleElementNames 属性指定其返回类型：

```
[return: TupleElementNames(new string[] {"result", "remainder" })]
public static ValueTuple<int, int> Divide(int dividend, int divisor)
```

当使用这种方式调用方法时，编译器会从属性中读取信息，以将名称与 ItemX 字段匹配。进行该调用时，将使用 ItemX 字段而不是更好的名称。

在 TupleElementNames 属性的自动用法中，返回元组的方法可以在库中声明(代码文件 TuplesLib/SimpleMath.cs)：

```
public class SimpleMath
{
  public static (int result, int remainder) Divide(int dividend, int divisor)
  {
    int result = dividend / divisor;
    int remainder = dividend % divisor;
    return (result, remainder);
  }
}
```

该库是在控制台应用程序中使用的，其 result 和 remainder 名称可以直接使用：

```
private static void UseALibrary()
{
  var t = SimpleMath.Divide(5, 3);
  Console.WriteLine($"result: {t.result}, remainder: {t.remainder}");
}
```

旧的 Tuple 类型是一个类，而新的元组 ValueTuple 是一个结构。这减少了把值类型存储在堆栈上时垃圾收集器所需的工作。旧的 Tuple 类型实现为一个具有只读属性的不可变类。在新的 ValueTuple 中，成员是公共字段。公共字段使这种类型可变(代码文件 Tuples/Program.cs)：

```
static void Mutability()
{
  // old tuple is a immutable reference type
  Tuple<string, int> t1 = Tuple.Create("old tuple", 42);
  // t1.Item1 = "new string"; // not possible with Tuple

  // new tuple is a mutable value type
  (string s, int i) t2 = ("new tuple", 42);
  t2.s = "new string";
  t2.i = 43;
  t2.i++;

  Console.WriteLine($"new string: {t2.s} int: {t2.i}");
}
```

**注意：**

微软似乎违反了 ValueTuple 的一些规则：结构应该是不可变的，而字段不应该声明为 public。但是，新的元组可以与简单值类型(例如 int 和 long)相媲美；使用元组打破规则是完全可原谅的，也可以获得最佳性能优化。

### 13.6.5　ValueTuple 与元组的兼容性

旧的元组类型由于命名不太好而用得不太多。但是，对于使用 Tuple 类型的程序，很容易将其转换为 ValueTuple。

调用 ToValueTuple 扩展方法可以将 Tuple 类型转换为 ValueTuple。由于旧的 Tuple 类型没有提供更好的名称，因此需要用圆括号定义名称(代码文件 Tuples /Program.cs)：

```
static void TupleCompatibility()
{
  // convert Tuple to ValueTuple
  Tuple<string, int, bool, Person> t1 = Tuple.Create("a string", 42, true,
    new Person("Katharina", "Nagel"));
  Console.WriteLine($"old tuple - string: {t1.Item1}, number: {t1.Item2},
    bool: {t1.Item3}, Person: {t1.Item4}");
  (string s, int i, bool b, Person p) t2 = t1.ToValueTuple();
  Console.WriteLine($"new tuple - string: {t2.s}, number: {t2.i}, bool: {t2.b},
    Person: {t2.p}");
  //...
}
```

旧元组也可以解构到特定的字段。下例显示了将元组 t1 解构到字段 s、i 和 b。

```
static void TupleCompatibility()
{
  //...
  (string s, int i, bool b, Person p) = t1; // Deconstruct
  Console.WriteLine($"new tuple - string: {s}, number: {i}, bool: {b},
    Person {p}");
  //...
}
```

反过来也是可能的。新的值元组可以用 ToTuple 方法转换成元组。当然，需要使用 Item1、Item2、Item3 等指定成员。

```
static void TupleCompatibility()
{
  //...
  // convert ValueTuple to Tuple
  Tuple<string, int, bool, Person> t3 = t2.ToTuple();
  Console.WriteLine($"old tuple - string: {t1.Item1}, number: {t1.Item2}, " +
    $"bool: {t1.Item3}, Person: {t1.Item4}");
}
```

### 13.6.6　推断出元组名称

C# 7.1 的一个新特性是推断元组的名称。前面声明的 Divide 方法返回一个包含名称 result 和 remainder 的元组。返回的元组写入变量 t1，其中这些名称用于访问元组字段。当第二次调用 Divide 方法时，将 tuple 结果写

入一个带有名称 res 和 rem 的元组。在返回的元组中，result 写入 res，remainder 写入 rem，t3 是使用一个元组字面量创建的，其中定义了 res 和 rem 字段，并相应地分配元组 t1 的值。本例中的第四个元组使用名称推断，t4 是使用一个元组字面量创建的，其中的名称与元组 t1 的名称相同。在不给元组成员提供名称的情况下访问 result 和 remainder，会使元组成员和 t1 中的字段同名，t4 也有名称为 result 和 remainder 的成员(代码文件 Tuples/Program.cs)：

```
private static void TupleNames()
{
  var t1 = Divide(9, 4);
  Console.WriteLine($"{t1.result}, {t1.remainder}");

  (int res, int rem) t2 = Divide(11, 3);
  Console.WriteLine($"{t2.res}, {t2.rem}");

  var t3 = (res: t1.result, rem: t1.remainder);

  // use inferred names
  var t4 = (t1.result, t1.remainder);
  Console.WriteLine($"{t4.result}, {t4.remainder}");
}
```

**注意：**

元组名称的推断至少需要 C# 7.1。需要在 csproj 项目文件中使用 LangVersion 指定这个版本，或者在 Visual Studio 中使用 Project Settings 指定它。

## 13.6.7　元组与链表

元组的实际使用与链表有关。链表中的一项(它是一个 LinkedListNode)包含了这个项的值和对下一项的引用。在下面的代码片段中，创建一个包含 10 个元素的链表。然后，使用 do/while 语句遍历这个列表。在循环中，使用元组字面量访问 LinkedListNode 的 Value 和 Next 属性。通过解构，将值写入变量 Value，链表中的下一项写入变量 node，该变量本身就是 LinkedListNode(代码文件 Tuples/Program.cs)：

```
static void TuplesWithLinkedList()
{
  Console.WriteLine(nameof(TuplesWithLinkedList));
  var list = new LinkedList<int>(Enumerable.Range(0, 10));

  int value;
  LinkedListNode<int> node = list.First;
  do
  {
    (value, node) = (node.Value, node.Next);
    Console.WriteLine(value);
  } while (node != null);
  Console.WriteLine();
}
```

**注意：**

链表在第 10 章中讨论。

## 13.6.8　元组和 LINQ

第 12 章使用 LINQ 语句演示了匿名类型和元组。下面将一个 LINQ 查询从匿名类型改为元组。下面的 LINQ 查询创建了一个匿名类型，并在 Select 方法的参数中指定了 LastName 和 Starts 属性(代码文件 Tuples/Program.cs)：

```
static void UsingAnonymousTypes()
{
  var racerNamesAndStarts = Formula1.GetChampions()
    .Where(r => r.Country == "Italy")
    .OrderByDescending(r => r.Wins)
    .Select(r => new
    {
      r.LastName,
      r.Starts
```

```
    });

  foreach (var r in racerNamesAndStarts)
  {
    Console.WriteLine($"{r.LastName}, starts: {r.Starts}");
  }
}
```

将花括号改为圆括号，创建一个带有字段 LastName 和 Starts 的元组。

```
static void UsingTuples()
{
  var racerNamesAndStarts = Formula1.GetChampions()
    .Where(r => r.Country == "Italy")
    .OrderByDescending(r => r.Wins)
    .Select(r =>
    (
      r.LastName,
      r.Starts
    ));

  foreach (var r in racerNamesAndStarts)
  {
    Console.WriteLine($"{r.LastName}, starts: {r.Starts}");
  }
}
```

**注意：**

对于匿名类型，创建一个类，因此该类的实例会分配到堆上，需要从垃圾收集器中收集。相比较而言，元组是值类型，存储在堆栈上。元组具有性能优势。

### 13.6.9　解构

前面介绍了元组的解构——将元组写入简单变量。解构也可以用任何类型来完成：把类或结构分解为它的各个部分。

例如，前面所示的 Person 类可以分解为姓和名(代码文件 Tuples /Program.cs)：

```
private static void Deconstruct()
{
  var p1 = new Person("Katharina", "Nagel");

  (var first, var last) = p1;
  Console.WriteLine($"{first} {last}");
}
```

只需要创建 Deconstruct 方法，将分离的部分放入 out 参数中(代码文件 Tuples /Program.cs)。

```
public class Person
{
  public Person(string firstName, string lastName)
  {
    FirstName = firstName;
    LastName = lastName;
  }
  public string FirstName { get; }
  public string LastName { get; }

  public override string ToString() => $"{FirstName} {LastName}";

  public void Deconstruct(out string firstName, out string lastName)
  {
    firstName = FirstName;
    lastName = LastName;
  }
}
```

解构是用方法名 Deconstruct 实现的。该方法总是 void 类型，并用 out 参数返回各个部分。为什么创建元组的方法不能通过返回元组来实现？原因是允许重载。可以使用不同的参数类型实现多个 Deconstruct 方法。这在返回元组时是不可能的。在 C#中，重载方法不能仅通过它的返回类型来区分。

### 13.6.10 解构与扩展方法

即使不给应解构的类添加 Deconstruct 方法，解构也是可以实现的：使用扩展方法。下面的代码示例为 Racer 类型定义了一个扩展方法，将 Racer 解构为 firstName、lastName、starts 和 wins(代码文件 Tuples/RacerExtensions.cs)：

```
public static class RacerExtensions
{
  public static void Deconstruct(this Racer r, out string firstName,
    out string lastName, out int starts, out int wins)
  {
    firstName = r.FirstName;
    lastName = r.LastName;
    starts = r.Starts;
    wins = r.Wins;
  }
}
```

下面的代码片段将 Racer 分解为变量 first和 last。Starts 和 wins 被忽略(代码文件 Tuples/Program.cs)：

```
static void DeconstructWithExtensionsMethods()
{
  var racer = Formula1.GetChampions().Where(
    r => r.LastName == "Lauda").First();
  (string first, string last, _, _) = racer;
  Console.WriteLine($"{first} {last}");
}
```

元组是 C# 7 中最重要的改进之一(如果不是最重要的话)。接下来学习模式匹配，这是 C# 7 的另一个重要特性。

## 13.7 模式匹配

从面向对象的观点来看，最好总是使用具体的类型和接口来解决问题。然而，通常这并不容易做到。在数据库中，查询可能会给出与任何层次结构都无关的不同对象类型。访问 API 服务时，可以返回一个列表或对象，或者可能什么也不返回。因此，方法通常应该与不同的类型一起工作。这就是模式匹配可以提供帮助的地方。

例如，下面创建了一个不同对象的数组。在这个名为 data 的数组中，第一个元素是 null，其后是值为 42 的整数、一个字符串、一个 Person 类型的对象，以及一个包含 Person 对象的数组(代码文件 PatternMatching/Program.cs)：

```
static void Main()
{
  var p1 = new Person("Katharina", "Nagel");
  var p2 = new Person("Matthias", "Nagel");
  var p3 = new Person("Stephanie", "Nagel");
  object[] data = { null, 42, "astring", p1, new Person[] { p2, p3 } };

  foreach (var item in data)
  {
    IsOperator(item);
  }

  foreach (var item in data)
  {
    SwitchStatement(item);
  }
}
```

在 C# 7 中的模式匹配中，is 运算符和 switch 语句得到了三种模式的增强：const 模式、type 模式和 var 模式。下面从 is 运算符开始详细介绍。

### 13.7.1 模式匹配与 is 运算符

与 is 运算符的简单匹配是 const 模式。在这个模式中，可以将对象与常量值进行比较，比如 null 或 42(代

码文件 PatternMatching/Program.cs)：

```
static void IsOperator(object item)
{
  // const pattern
  if (item is null)
  {
    Console.WriteLine("item is null");
  }

  if (item is 42)
  {
    Console.WriteLine("item is 42");
  }
  //...
}
```

使用前面声明的数组运行应用程序时，数组的前两项与两个 if 语句相匹配，如下面的程序输出所示：

```
item is null
item is 42
```

**注意：**

方法的参数通常要检查是否为 null，一般是使用相等运算符来比较 null。例如：

```
if (item == null) throw ArgumentNullException("null");
```

现在可以使用模式匹配替换：

```
if (item is null) throw ArgumentNullException("null");
```

在幕后，C#编译器生成相同的中间语言(IL)代码。

最有趣的模式匹配是 type 模式。使用此模式，可以匹配特定的类型，例如 int 或 string。该模式还允许声明变量，例如 if(itemisinti)，如果该模式适用，则将变量 i 分配给该项：

```
static void IsOperator(object item)
{
  //...
  // type pattern
  if (item is int)
  {
    Console.WriteLine($"Item is of type int");
  }

  if (item is int i)
  {
    Console.WriteLine($"Item is of type int with the value {i}");
  }

  if (item is string s)
  {
    Console.WriteLine($"Item is a string: {s}");
  }
  //...
}
```

对于前面的类型模式，这些匹配应用于值 4 和字符串 astring。

```
Item is of type int
Item is of type int with a value 42
Item is a string: astring
```

声明某类型的一个变量允许强类型化的访问。可以访问该类型的所有成员，而不需要进行类型转换。这也允许在 if 语句中使用逻辑运算符来检查其他约束，而不仅仅是类型，比如 FirstName 以字符串 Ka 开头：

```
static void IsOperator(object item)
{
  //...
  if (item is Person p && p.FirstName.StartsWith("Ka"))
  {
    Console.WriteLine($"Item is a person: {p.FirstName} {p.LastName}");
  }

  if (item is IEnumerable<Person> people)
```

```
    {
      string names = string.Join(", ",
        people.Select(p1 => p1.FirstName).ToArray());
      Console.WriteLine($"it's a Person collection containing {names}");
    }
    //...
  }
```

使用前两种类型模式和应用的对象数组，这些匹配应用于：

```
Item is a person: Katharina Nagel
it's a Person collection containing Matthias, Stephanie
```

还需要讨论一个模式类型：var 模式。一切都可以应用于 var；只需要得到具体的类型。在样例代码中，将调用 GetType 方法来获取类型的名称，并将具体类型写入控制台。当值为 null 时，var 模式也适用。这就是为什么 every 变量使用 null 条件运算符的原因。如果项为 null，则 every 为空，该项将字符串 null 写入控制台：

```
static void IsOperator(object item)
{
  //...
  // var pattern
  if (item is var every)
  {
    Console.WriteLine($"it's var of type {every?.GetType().Name ?? "null"} " +
      $"with the value {every ?? "nothing"}");
  }
}
```

var 模式的应用程序的输出表明，数组的所有项都与此模式匹配：

```
it's var of type null with the value nothing
it's var of type Int32 with the value 42
it's var of type String with the value astring
it's var of type Person with the value Katharina Nagel
it's var of type Person[] with the value PatternMatching.Person[]
```

### 13.7.2 模式匹配与 switch 语句

对于 switch 语句，也可以使用三种模式类型。下面的代码片段显示了用于 null 和 42 的 const 模式；用于 int、string 和 Person 的类型模式；以及 var 模式。与 is 运算符的扩展一样，对于 switch 语句，可以用类型模式指定变量，将匹配结果写入该变量。还可以在 when 子句中应用一个附加的过滤器。Person 类的第一个类型匹配仅适用于 Person 的 FirstName 属性值是 Katharina 的情形。在 switch 语句中，case 语句的顺序非常重要。一旦应用了一个 case，就不进一步检查其他 case。如果通过 when 子句应用了第一个匹配的 Person 类型，就不应用对 Person 的第二个 case。这就是为什么在对类型处理一般 case 之前，必须先进行 when 过滤。用最后一个 case 中定义的 var 模式与传递给 switch 语句的每个对象匹配。但是，只有没有应用前面定义的其他 case 时，才会检查这个 case。default 子句可以在 switch 语句的每个位置上，只有在没有匹配的 case 时才适用。最好把这个子句放在最后(代码文件 PatternMatching/Program.cs)：

```
static void SwitchStatement(object item)
{
  switch (item)
  {
    case null:
    case 42:
      Console.WriteLine("it's a const pattern");
      break;
    case int i:
      Console.WriteLine($"it's a type pattern with int: {i}");
      break;
    case string s:
      Console.WriteLine($"it's a type pattern with string: {s}");
      break;
    case Person p when p.FirstName == "Katharina":
      Console.WriteLine($"type pattern match with Person and " +
        $"when clause: {p}");
      break;
    case Person p:
      Console.WriteLine($"type pattern match with Person: {p}");
      break;
```

```
        case var every:
          Console.WriteLine($"var pattern match: {every?.GetType().Name}");
          break;
        default:
      }
    }
```

运行应用程序时，switch 语句的 const 模式适用于 null 和 42，字符串模式适用于字符串 astring,第一个 Person case 应用于 Person 对象，最后，Person 数组与 var 模式匹配，因为之前没有应用其他模式。没有与 int 类型匹配的类型，因为 const 模式匹配较早：

```
it's a const pattern
it's a const pattern
it's a type pattern with string: astring
type pattern match with Person and when clause: Katharina Nagel
var pattern match: Person[]
```

### 13.7.3　模式匹配与泛型

如果需要与泛型相匹配的模式，则需要将编译器配置为至少 C# 7.1。C# 7.1 为泛型添加了模式匹配。使用 C# 7，可以定义一个泛型方法，并使用 is 运算符检查泛型类型的变量，以应用于特定的类型（代码文件 PatternMatching/HttpManager.cs）：

```
public void Send<T>(T package)
{
  if (package is HealthPackage hp)
  {
    hp.CheckHealth();
  }
  //...
}
```

可以与泛型进行模式匹配，类似于使用泛型类的方式。还可以对泛型使用模式匹配和 switch 语句。

> **注意：**
> 第 5 章讨论了泛型方法和泛型类。

## 13.8　小结

本章介绍了 C# 7 的新特性，比如本地函数、元组和模式匹配。所有这些特性都来自函数式编程范式，但是对于创建普通的.NET 应用程序来说，这些都非常有用。本地函数在一些场景中是有用的，比如允许使用 yield 语句来处理延迟方法的更好的错误处理方式。元组提供了一种组合不同数据类型的有效方法。不必总是为这样的组合创建自定义类。

本章还讨论了在 LINQ 查询中元组如何替换匿名类型。模式匹配允许使用 is 运算符和 switch 语句的增强来处理不同的类型。

下一章将讨论错误和异常的细节。

# 第 14 章

# 错误和异常

**本章要点**

- 异常类
- 使用 try…catch…finally 捕获异常
- 过滤异常
- 创建用户定义的异常
- 获取调用者的信息

**本章源代码下载地址(wrox.com)：**

打开 www.wrox.com 的 Download Code 选项卡可下载本章源代码。源代码也可以在 ErrorsAndExceptions 目录的 https://github.com/ProfessionalCSharp/ProfessionalCSharp7 中找到。本章代码分为以下几个主要的示例文件：

- Simple Exceptions
- ExceptionFilters
- RethrowExceptions
- SolicitColdCall
- CallerInformation

## 14.1 简介

错误的出现并不总是编写应用程序的人的原因，有时应用程序会因为应用程序的最终用户引发的动作或运行代码的环境而发生错误。无论如何，我们都应预测应用程序中出现的错误，并相应地进行编码。

.NET Framework 改进了处理错误的方式。C#处理错误的机制可以为每种错误提供自定义处理方式，并把识别错误的代码与处理错误的代码分离开来。

无论编码技术有多好，程序都必须能处理可能出现的任何错误。例如，在一些复杂的代码处理过程中，代码没有读取文件的许可，或者在发送网络请求时，网络可能会中断。在这种异常情况下，方法只返回相应的错误代码通常是不够的——可能方法调用嵌套了 15 级或者 20 级，此时，程序需要跳过所有的 15 或 20 级方法调用，才能完全退出任务，并采取相应的应对措施。C#语言提供了处理这种情形的最佳工具，称为异常处理机制。

本章介绍了在多种不同的场景中捕获和抛出异常的方式。讨论不同名称空间中定义的异常类型及其层次结

构，并学习如何创建自定义异常类型。还将学到捕获异常的不同方式，例如，捕获特定类型的异常或者捕获基类的异常。本章还会介绍如何处理嵌套的 try 块，以及如何以这种方式捕获异常。对于无论如何都要调用的代码——即使发生了异常或者代码带错运行，可以使用本章介绍的 try/finally 块。

学习完本章后，你将很好地掌握 C#应用程序中的高级异常处理技术。

## 14.2　异常类

在 C#中，当出现某个特殊的异常错误条件时，就会创建(或抛出)一个异常对象。这个对象包含有助于跟踪问题的信息。我们可以创建自己的异常类(详见后面的内容)，但.NET 提供了许多预定义的异常类，多到这里不可能提供详尽的列表。在图 14-1 类的层次结构图中显示了其中的一些类，它们给出了大致的模式。本节将简要介绍在.NET 基类库中可用的一些异常。

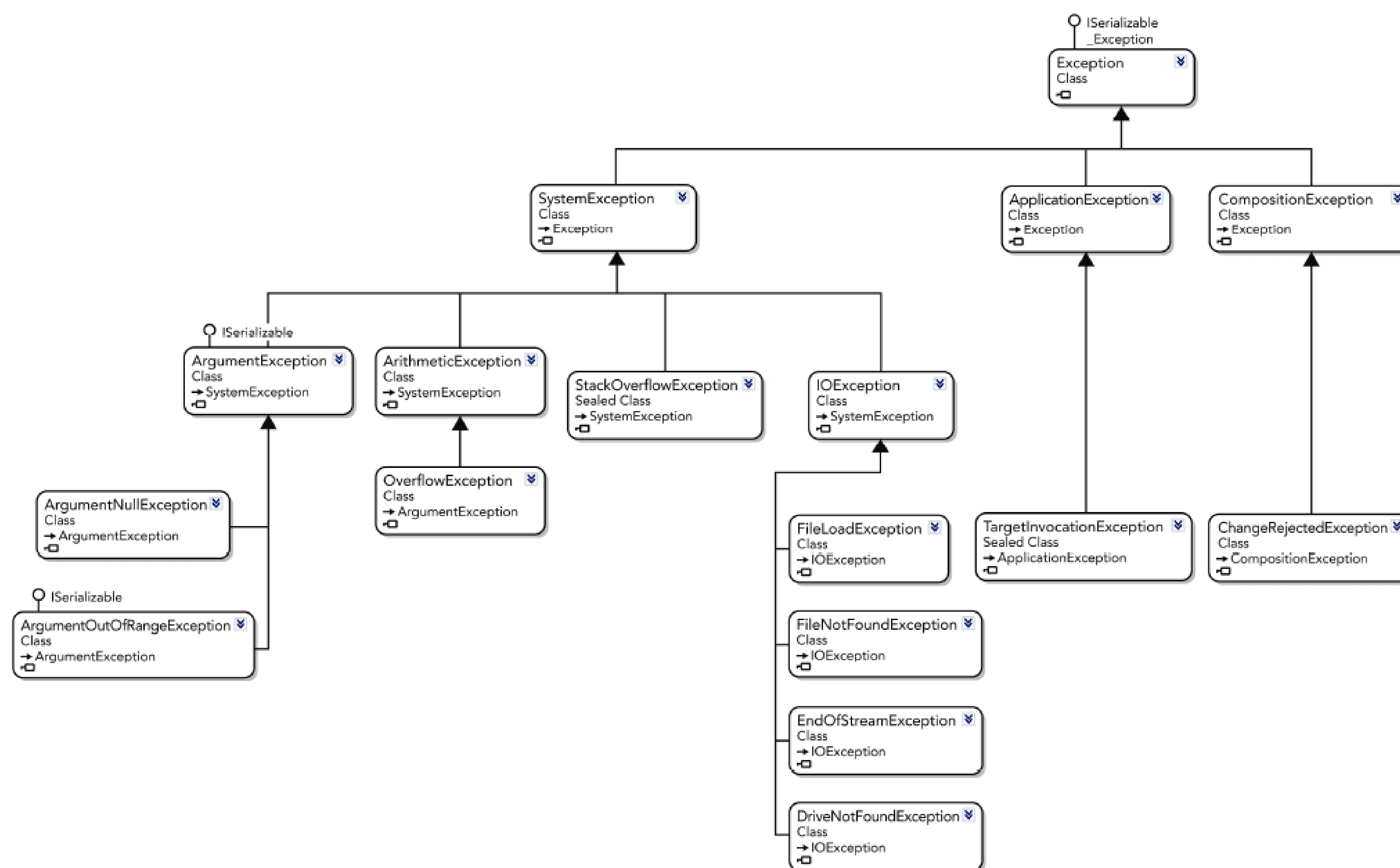



图 14-1

图 14-1 中的所有类都在 System 名称空间中，但 IOException 类、CompositionException 类和派生于这两个类的类除外。IOException 类及其派生类在 System.IO 名称空间中。System.IO 名称空间处理文件数据的读写。CompositionException 及其派生类在 System.ComponentModel.Composition 名称空间中。该名称空间处理部件和组件的动态加载。一般情况下，异常没有特定的名称空间，异常类应放在生成异常的类所在的名称空间中，因此与 IO 相关的异常就在 System.IO 名称空间中。在许多基类名称空间中都有异常类。

对于.NET 类，一般的异常类 System.Exception 派生自 System.Object，通常不在代码中抛出 System.Exception 泛型对象，因为它们无法确定错误情况的本质。

在该层次结构中有两个重要的类，它们派生自 System.Exception 类：

- SystemException——该类用于通常由.NET 运行库抛出的异常，或者由几乎所有的应用程序抛出的异常。例如，如果.NET 运行库检测到栈已满，它就会抛出 StackOverflowException 异常。另一方面，如果检测到调用方法时参数不正确，就可以在自己的代码中选择抛出 ArgumentException 异常或其子类异常。SystemException 异常的子类包括表示致命错误和非致命错误的异常。

- ApplicationException——在.NET Framework 最初的设计中，是打算把这个类作为自定义应用程序异常类的基类的。不过，CLR 抛出的一些异常类也派生自这个类(例如，TargetInvocationException)，应用程序抛出的异常则派生自 SystemException(例如，ArgumentException)。因此从 ApplicationException 派生自定义异常类型没有提供任何好处，所以不再是一种好做法。取而代之的是，可以直接从 Exception 基类派生自定义异常类。.NET Framework 中的许多异常类直接派生自 Exception。

其他可能用到的异常类包括：

- StackOverflowException——如果分配给栈的内存区域已满，就会抛出这个异常。如果一个方法连续地递归调用它自己，就可能发生栈溢出。这一般是一个致命错误，因为它禁止应用程序执行除了中断以外的其他任务。在这种情况下，甚至也不可能执行 finally 块。通常用户自己不能处理像这样的错误，而应退出应用程序。
- EndOfStreamException——这个异常通常是因为读到文件末尾而抛出的。流表示数据源之间的数据流。流详见第 23 章。
- OverflowException——如果要在 checked 上下文中把包含值–40 的 int 类型数据强制转换为 uint 数据，就会抛出这个异常。

我们不打算讨论图 14-1 中的其他异常类。显示它们仅为了演示异常类的层次结构。

异常类的层次结构并不多见，因为其中的大多数类并没有给它们的基类添加任何功能。但是在处理异常时，添加继承类的一般原因是更准确地指定错误条件，所以不需要重写方法或添加新方法(尽管常常要添加额外的属性，以包含有关错误情况的额外信息)。例如，当传递了不正确的参数值时，可给方法调用使用 ArgumentException 基类，ArgumentNullException 类派生于 ArgumentException 异常类，它专门用于处理所传递的参数值是 Null 的情况。

## 14.3 捕获异常

.NET 提供了大量的预定义基类异常对象，本节介绍如何在代码中使用它们捕获错误情况。为了在 C#代码中处理可能的错误情况，一般要把程序的相关部分分成 3 种不同类型的代码块：

- try 块包含的代码组成了程序的正常操作部分，但这部分程序可能遇到某些严重的错误。
- catch 块包含的代码处理各种错误情况，这些错误是执行 try 块中的代码时遇到的。这个块还可以用于记录错误。
- finally 块包含的代码清理资源或执行通常要在 try 块或 catch 块末尾执行的其他操作。无论是否抛出异常，都会执行 finally 块，理解这一点非常重要。因为 finally 块包含了应总是执行的清理代码，如果在 finally 块中放置了 return 语句，编译器就会标记一个错误。例如，使用 finally 块时，可以关闭在 try 块中打开的连接。finally 块是完全可选的。如果不需要清理代码(如删除对象或关闭已打开的对象)，就不需要包含此块。

下面的步骤说明了这些块是如何组合在一起捕获错误情况的：

(1) 执行的程序流进入 try 块。

(2) 如果在 try 块中没有错误发生，在块中就会正常执行操作。当程序流到达 try 块末尾后，如果存在一个 finally 块，程序流就会自动进入 finally 块(第(5)步)。但如果在 try 块中程序流检测到一个错误，程序流就会跳转到 catch 块(第(3)步)。

(3) 在 catch 块中处理错误。

(4) 在 catch 块执行完后，如果存在一个 finally 块，程序流就会自动进入 finally 块：

(5) 执行 finally 块(如果存在)。

用于完成这些任务的 C#语法如下所示：

```
try
{
  // code for normal execution
```

```
}
catch
{
  // error handling
}
finally
{
  // clean up
}
```

实际上，上面的代码还有几种变体：

- 可以省略 finally 块，因为它是可选的。
- 可以提供任意多个 catch 块，处理不同类型的错误。但不应包含过多的 catch 块，以防降低应用程序的性能。
- 可以定义过滤器，其中包含的 catch 块仅在过滤器匹配时，捕获特定块中的异常。
- 可以省略 catch 块。此时，该语法不是标识异常，而是一种确保程序流在离开 try 块后执行 finally 块中的代码的方式。如果在 try 块中有几个出口点，这很有用。

这看起来很不错，实际上是有问题的。如果运行 try 块中的代码，则程序流如何在错误发生时切换到 catch 块？如果检测到一个错误，代码就执行一定的操作，称为"抛出一个异常"；换言之，它实例化一个异常对象类，并抛出这个异常：

```
throw new OverflowException();
```

这里实例化了 OverflowException 类的一个异常对象。只要应用程序在 try 块中遇到一条 throw 语句，就会立即查找与这个 try 块对应的 catch 块。如果有多个与 try 块对应的 catch 块，应用程序就会查找与 catch 块对应的异常类，确定正确的 catch 块。例如，当抛出一个 OverflowException 异常对象时，执行的程序流就会跳转到下面的 catch 块：

```
catch (OverflowException ex)
{
  // exception handling here
}
```

换言之，应用程序查找的 catch 块应表示同一个类(或基类)中匹配的异常类实例。

有了这些额外的信息，就可以扩展刚才介绍的 try 块。为了讨论方便，假定可能在 try 块中发生两个严重错误：溢出和数组超出范围。假定代码包含两个布尔变量 Overflow 和 OutOfBounds，它们分别表示这两种错误情况是否存在。我们知道，存在表示溢出的预定义溢出异常类 OverflowException；同样，存在预定义的 IndexOutOfRangeException 异常类，用于处理超出范围的数组。

现在，try 块如下所示：

```
try
{
  // code for normal execution
  if (Overflow == true)
  {
    throw new OverflowException();
  }
  // more processing
  if (OutOfBounds == true)
  {
    throw new IndexOutOfRangeException();
  }
  // otherwise continue normal execution
}
catch (OverflowException ex)
{
  // error handling for the overflow error condition
}
catch (IndexOutOfRangeException ex)
{
  // error handling for the index out of range error condition
}
finally
{
  // clean up
}
```

这是因为 throw 语句可以嵌套在 try 块的几个方法调用中，甚至在程序流进入其他方法时，也会继续执行同一个 try 块。如果应用程序遇到一条 throw 语句，就会立即退出栈上所有的方法调用，查找 try 块的结尾和合适的 catch 块的开头，此时，中间方法调用中的所有局部变量都会超出作用域。try…catch 结构最适合于本节开头描述的场合：错误发生在一个方法调用中，而该方法调用可能嵌套了 15 或 20 级，这些处理操作会立即停止。

从上面的论述可以看出，try 块在控制执行的程序流上有重要的作用。但是，异常是用于处理异常情况的，这是其名称的由来。不应该用异常来控制退出 do…while 循环的时间。

### 14.3.1 异常和性能

异常处理具有性能含义。在常见的情况下，不应该使用异常处理错误。例如，将字符串转换为数字时，可以使用 int 类型的 Parse 方法。如果传递给此方法的字符串不能转换为数字，则此方法抛出 FormatException 异常；如果可以转换一个数字，但它不能放在 int 类型中，则抛出 OverflowException 异常：

```
static void NumberDemo1(string n)
{
  if (n is null) throw new ArgumentNullException(nameof(n));
  try
  {
    int i = int.Parse(n);
    Console.WriteLine($"converted: {i}");
  }
  catch (FormatException ex)
  {
    Console.WriteLine(ex.Message);
  }
  catch (OverflowException ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

如果 NumberDemo1 方法通常只用于在字符串中传递数字而接收不到数字是异常的，那么可以这样编写它。但是，如果在程序流的正常情况下，期望的字符串不能转换时，可以使用 TryParse 方法。如果字符串不能转换为数字，此方法不会抛出异常。相反，如果解析成功，TryParse 返回 true；如果解析失败，则返回 false：

```
static void NumberDemo2(string n)
{
  if (n is null) throw new ArgumentNullException(nameof(n));
  if (int.TryParse(n, out int result))
  {
    Console.WriteLine($"converted {result}");
  }
  else
  {
    Console.WriteLine("not a number");
  }
}
```

### 14.3.2 实现多个 catch 块

要了解 try…catch…finally 块是如何工作的，最简单的方式是用两个示例来说明。第一个示例是 SimpleExceptions。它多次要求用户输入一个数字，然后显示这个数字。为了便于解释这个示例，假定该数字必须在 0 到 5 之间，否则程序就不能对该数字进行正确的处理。所以，如果用户输入超出该范围的数字，程序就抛出一个异常。程序会继续要求用户输入更多数字，直到用户不再输入任何内容，按回车键为止。

**注意：**
这段代码没有说明何时使用异常处理，但是它显示了使用异常处理的好方法。顾名思义，异常用于处理异常情况。用户经常输入一些无聊的东西，所以这种情况不会真正发生。正常情况下，程序会处理不正确的用户输入，方法是进行即时检查，如果有问题，就要求用户重新输入。但是，在一个要求几分钟内读懂的小示例中生成异常是比较困难的，为了描述异常是如何工作的，后面将使用更真实的示例。

SimpleExceptions 的代码如下所示(代码文件 SimpleExceptions/Program.cs)：

```
public class Program
{
  public static void Main()
  {
    while (true)
    {
      try
      {
        string userInput;
        Console.Write("Input a number between 0 and 5 " +
          "(or just hit return to exit)> ");
        userInput = Console.ReadLine();

        if (string.IsNullOrEmpty(userInput))
        {
          break;
        }
        int index = Convert.ToInt32(userInput);
        if (index < 0 || index > 5)
        {
          throw new IndexOutOfRangeException($"You typed in {userInput}");
        }
        Console.WriteLine($"Your number was {index}");
      }
      catch (IndexOutOfRangeException ex)
      {
        Console.WriteLine("Exception: " +
          $"Number should be between 0 and 5. {ex.Message}");
      }
      catch (Exception ex)
      {
        Console.WriteLine($"An exception was thrown. Message was: " +
          $"{ex.Message}");
      }
      finally
      {
        Console.WriteLine("Thank you\n");
      }
    }
  }
}
```

这段代码的核心是一个 while 循环，它连续使用 ReadLine()方法以请求用户输入。ReadLine()方法返回一个字符串，所以程序首先要用 System.Convert.ToInt32()方法把它转换为 int 型。System.Convert 类包含执行数据转换的各种有用方法，并提供了 int.Parse()方法的一个替代方法。一般情况下，System.Convert 类包含执行各种类型转换的方法，C#编译器把 int 解析为 System.Int32 基类的实例。

**注意：**
值得注意的是，传递给 catch 块的参数只能用于该 catch 块。这就是为什么在上面的代码中，能在后续的 catch 块中使用相同的参数名 ex 的原因。

在上面的代码中，我们也检查一个空字符串，因为该空字符串是退出 while 循环的条件。注意这里用 break 语句退出 try 块和 while 循环——这是有效的。当然，当程序流退出 try 块时，会执行 finally 块中的 Console.WriteLine()语句。尽管这里仅显示一句问候，但一般在这里可以关闭文件句柄，调用各种对象的 Dispose()方法，以执行清理工作。一旦应用程序退出了 finally 块，它就会继续执行下一条语句，如果没有 finally 块，该语句也会执行。在本例中，我们返回到 while 循环的开头，再次进入 try 块(除非执行 while 循环中 break 语句的结果是进入 finally 块，此时就会退出 while 循环)。

下面看看异常情况：

```
if (index < 0 || index > 5)
{
  throw new IndexOutOfRangeException($"You typed in {userInput}");
}
```

在抛出一个异常时，需要选择要抛出的异常类型。可以使用 System.Exception 异常类，但这个类是一个基类，

最好不要把这个类的实例当作一个异常抛出，因为它没有包含关于错误的任何信息。而.NET Framework 包含了许多派生自 System.Exception 异常类的其他异常类，每个类都对应于一种特定类型的异常情况，也可以定义自己的异常类。在抛出一个匹配特定错误情况的类的实例时，应提供尽可能多的异常信息。在前面的例子中，System.IndexOutOfRangeException 异常类是最佳选择。IndexOutOfRangeException 异常类有几个重载的构造函数，我们选择的一个重载，其参数是一个描述错误的字符串。另外，也可以选择派生自己的自定义异常对象，它描述该应用程序环境中的错误情况。

假定用户这次输入了一个不在 0～5 范围内的数字，if 语句就会检测到一个错误，并实例化和抛出一个 IndexOutOfRangeException 异常对象。应用程序会立即退出 try 块，并查找处理 IndexOutOf RangeException 异常的 catch 块。它遇到的第一个 catch 块如下所示：

```
catch (IndexOutOfRangeException ex)
{
  Console.WriteLine($"Exception: Number should be between 0 and 5." +
    $"{ex.Message}");
}
```

由于这个 catch 块带合适类的一个参数，因此它会接收异常实例，并被执行。在本例中，是显示错误信息和 Exception.Message 属性(它对应于传递给 IndexOutofRangeException 的构造函数的字符串)。执行了这个 catch 块后，控制权就切换到 finally 块，就好像没有发生过任何异常。

注意，本示例还提供了另一个 catch 块：

```
catch (Exception ex)
{
  Console.WriteLine($"An exception was thrown. Message was: {ex.Message}");
}
```

如果没有在前面的 catch 块中捕获到这类异常，则这个 catch 块也能处理 IndexOutOfRangeException 异常。基类的一个引用也可以指向派生自它的类的所有实例，所有的异常都派生自 Exception 类。这个 catch 块没有执行，因为应用程序只执行它在可用的 catch 块列表中找到的第一个合适的 catch 块。第二个 catch 块捕获派生自 Exception 基类的其他异常。请注意在 try 块中对方法(Console.ReadLine、Console.Write 和 Convert.ToInt32)的三个单独调用，可能会抛出其他异常。

如果输入的内容不是数字，如 a 或 hello，则 Convert.ToInt32()方法就会抛出 System.FormatException 类的一个异常，表示传递给 ToInt32()方法的字符串对应的格式不能转换为 int。此时，应用程序会跟踪这个方法调用，查找可以处理该异常的处理程序。第一个 catch 块带一个 IndexOutOfRangeException 异常，不能处理这种异常。应用程序接着查看第二个 catch 块，显然它可以处理这类异常，因为 FormatException 异常类派生于 Exception 异常类，所以把 FormatException 异常类的实例作为参数传递给它。

该示例的这种结构是非常典型的多 catch 块结构。最先编写的 catch 块用于处理非常特殊的错误情况，接着是比较一般的块，它们可以处理任何错误，我们没有为它们编写特定的错误处理程序。实际上，catch 块的顺序很重要，如果以相反的顺序编写这两个块，代码就不会编译，因为第二个 catch 块是不会执行的(Exception catch 块会捕获所有异常)。因此，最上面的 catch 块应用于最特殊的异常情况，最后是最一般的 catch 块。

前面分析了该示例的代码，现在可以运行它。下面的输出说明了不同的输入会得到不同的结果，并说明抛出了 IndexOutOfRangeException 异常和 FormatException 异常：

```
SimpleExceptions
Input a number between 0 and 5 (or just hit return to exit)> 4
Your number was 4
Thank you
Input a number between 0 and 5 (or just hit return to exit)> 0
Your number was 0
Thank you
Input a number between 0 and 5 (or just hit return to exit)> 10
Exception: Number should be between 0 and 5. You typed in 10
Thank you
Input a number between 0 and 5 (or just hit return to exit)> hello
An exception was thrown. Message was: Input string was not in a correct format.
Thank you
Input a number between 0 and 5 (or just hit return to exit)>
Thank you
```

### 14.3.3　在其他代码中捕获异常

上面的示例说明了两个异常的处理。一个是 IndexOutOfRangeException 异常，它由我们自己的代码抛出，另一个是 FormatException 异常，它由一个基类抛出。如果检测到错误，或者某个方法因传递的参数有误而被错误调用，库中的代码就常常会抛出一个异常。但库中的代码很少捕获这样的异常。应由客户端代码来决定如何处理这些问题。

在调试时，异常经常从基类库中抛出，调试的过程在某种程度上是确定异常抛出的原因，并消除导致错误发生的缘由。主要目标是确保代码在发布后，异常只发生在非常少见的情况下，如果可能，应在代码中以适当的方式处理它。

### 14.3.4　System.Exception 属性

本示例只使用了异常对象的一个 Message 属性。在 System.Exception 异常类中还有许多其他属性，如表 14-1 所示。

表 14-1

| 属　性 | 说　明 |
|---|---|
| Data | 这个属性可以给异常添加键/值语句，以提供关于异常的额外信息 |
| HelpLink | 链接到一个帮助文件上，以提供关于该异常的更多信息 |
| InnerException | 如果此异常是在 catch 块中抛出的，它就会包含把代码发送到 catch 块中的异常对象 |
| Message | 描述错误情况的文本 |
| Source | 导致异常的应用程序名或对象名 |
| StackTrace | 栈上方法调用的详细信息，它有助于跟踪抛出异常的方法 |
| HResult | 分配给异常的一个数值 |
| TargetSite | .NET 反射对象，描述了抛出异常的方法 |

在这些属性中，如果可以进行栈跟踪，则 StackTrace 的属性值由.NET 运行库自动提供。Source 属性总是由.NET 运行库填充为抛出异常的程序集的名称(但可以在代码中修改该属性，提供更具体的信息)，Data、Message、HelpLink 和 InnerException 属性必须在抛出异常的代码中填充，方法是在抛出异常前设置这些属性。例如，抛出异常的代码如下所示：

```
if (ErrorCondition == true)
{
  var myException = new ClassMyException("Help!!!!");
  myException.Source = "My Application Name";
  myException.HelpLink = "MyHelpFile.txt";
  myException.Data["ErrorDate"] = DateTime.Now;
  myException.Data.Add("AdditionalInfo", "Contact Bill from the Blue   Team");
  throw myException;
}
```

其中，ClassMyException 是抛出的异常类的名称。注意所有异常类的名称通常以 Exception 结尾。另外，Data 属性可以用两种方式设置。

### 14.3.5　异常过滤器

自从 C# 6 开始就支持异常过滤器。catck 块仅在过滤器返回 true 时执行。捕获不同的异常类型时，可以有行为不同的 catch 块。在某些情况下，catch 块基于异常的内容执行不同的操作。例如，使用 Windows 运行库时，所有不同类型的异常通常都会得到 COM 异常，在执行网络调用时，许多不同的场景都会得到网络异常。例如，如果服务器不可用，或提供的数据不符合期望，以不同的方式应对这些错误是好事。一些异常可以用不同的方

式恢复，而在另外一些异常中，用户可能需要一些信息。

下面的代码示例抛出类型 MyCustomException 的异常，设置这个异常的 ErrorCode 属性(代码文件 ExceptionFilters/Program.cs)：

```
public static void ThrowWithErrorCode(int code)
{
  throw new MyCustomException("Error in Foo") { ErrorCode = code };
}
```

在 Main()方法中，try 块和两个 catch 块保护方法调用。第一个 catch 块使用 when 关键字过滤出 ErrorCode 属性等于 405 的异常。when 子句的表达式需要返回一个布尔值。如果结果是 true，这个 catch 块就处理异常。如果它是 false，就寻找其他 catch 块。给 ThrowWithErrorCode()方法传递 405，过滤器就返回 true，第一个 catch 块处理异常。传递另一个值，过滤器就返回 false，第二个 catch 块处理异常。使用过滤器，可以使用多个处理程序来处理相同的异常类型。

当然也可以删除第二个 catch 块，此时就不处理该情形下出现的异常。

```
try
{
  ThrowWithErrorCode(405);
}
catch (MyCustomException ex) when (ex.ErrorCode == 405)
{
  Console.WriteLine($"Exception caught with filter {ex.Message} " +
    $"and {ex.ErrorCode}");
}
catch (MyCustomException ex)
{
  Console.WriteLine($"Exception caught {ex.Message} and {ex.ErrorCode}");
}
```

### 14.3.6　重新抛出异常

捕获异常时，重新抛出异常也是非常普遍的。再次抛出异常时，可以改变异常的类型。这样，就可以给调用程序提供所发生的更多信息。原始异常可能没有上下文的足够信息。还可以记录异常信息，并给调用程序提供不同的信息。例如，为了让用户运行应用程序，异常信息并没有真正的帮助。阅读日志文件的系统管理员可以做出相应的反应。

重新抛出异常的一个问题是，调用程序往往需要通过以前的异常找出其发生的原因和地点。根据异常的抛出方式，堆栈跟踪信息可能会丢失。为了看到重新抛出异常的不同选项，示例程序 RethrowExceptions 显示了不同的选项。

对于此示例，创建了两个自定义的异常类型。第一个是 MyCustomException，除了基类 Exception 的成员之外，定义了属性 ErrorCode，第二个是 AnotherCustomException，支持传递一个内部异常(代码文件 RethrowExceptions /MyCustomException.cs)：

```
public class MyCustomException : Exception
{
  public MyCustomException(string message)
    : base(message) { }

  public int ErrorCode { get; set; }
}

public class AnotherCustomException : Exception
{
  public AnotherCustomException(string message, Exception innerException)
    : base(message, innerException) { }
}
```

HandleAll()方法调用 HandleAndThrowAgain、HandleAndThrowWithInnerException、HandleAnd Rethrow()和 HandleWithFilter()方法。捕获抛出的异常，把异常消息和堆栈跟踪写到控制台。为了更好地从堆栈跟踪中找到所引用的行号，使用预处理器指令#line，重新编号。这样，采用委托 m 调用的方法就在 114 行(代码文件 RethrowExceptions/Program.cs)：

```
#line 100
public static void HandleAll()
{
  var methods = new Action[]
  {
    HandleAndThrowAgain,
    HandleAndThrowWithInnerException,
    HandleAndRethrow,
    HandleWithFilter
  };

  foreach (var m in methods)
  {
    try
    {
      m(); // line 114
    }
    catch (Exception ex)
    {
      Console.WriteLine(ex.Message);
      Console.WriteLine(ex.StackTrace);
      if (ex.InnerException != null)
      {
        Console.WriteLine($"\tInner Exception{ex.Message}");
        Console.WriteLine(ex.InnerException.StackTrace);
      }
      Console.WriteLine();
    }
  }
}
```

ThrowAnException 方法用于抛出第一个异常。这个异常在 8002 行抛出。在开发期间，它有助于知道这个异常在哪里抛出：

```
#line 8000
public static void ThrowAnException(string message)
{
  throw new MyCustomException(message); // line 8002
}
```

### 1. 重新抛出异常的用法

方法 HandleAndThrowAgain 只是把异常记录到控制台，并使用 throw ex 再次抛出它：

```
#line 4000
public static void HandleAndThrowAgain()
{
  try
  {
    ThrowAnException("test 1");
  }
  catch (Exception ex)
  {
    Console.WriteLine($"Log exception {ex.Message} and throw again");
    throw ex; // you shouldn't do that - line 4009
  }
}
```

运行应用程序，简化的输出是显示堆栈跟踪(代码文件没有名称空间和完整的路径)，代码如下：

```
Log exception test 1 and throw again
test 1
at Program.HandleAndThrowAgain() in Program.cs:line 4009
at Program.HandleAll() in Program.cs:line 114
```

堆栈跟踪显示了在 HandleAll 方法中调用 m()方法，进而调用 HandleAndThrowAgain()方法。异常最初在哪里抛出的信息完全丢失在最后一个 catch 的调用堆栈中。于是很难找到错误的初始原因。通常不要传送异常对象，使用 throw 抛出同一个异常。

### 2. 改变异常

一个有用的场景是改变异常的类型，并添加错误信息。这在 HandleAndThrowWithInnerException()方法中完成。记录错误之后，抛出一个新的异常类型 AnotherException，传递 ex 作为内部异常：

```
#line 3000
public static void HandleAndThrowWithInnerException()
{
  try
  {
    ThrowAnException("test 2"); // line 3004
  }
  catch (Exception ex)
  {
    Console.WriteLine($"Log exception {ex.Message} and throw again");
    throw new AnotherCustomException("throw with inner exception", ex); // 3009
  }
}
```

检查外部异常的堆栈跟踪，会看到行号 3009 和 114，与前面相似。然而，内部异常给出了错误的最初原因。它给出调用了错误方法的行号(3004)和抛出最初(内部)异常的行号(8002)：

```
Log exception test 2 and throw again
throw with inner exception
at Program.HandleAndThrowWithInnerException() in Program.cs:line 3009
at Program.HandleAll() in Program.cs:line 114
Inner Exception throw with inner exception
at Program.ThrowAnException(String message) in Program.cs:line 8002
at Program.HandleAndThrowWithInnerException() in Program.cs:line 3004
```

这样不会丢失信息。

**注意：**

试图找到错误的原因时，看看内部异常是否存在。这往往会提供有用的信息。

**注意：**

捕获异常时，最好在重新抛出时改变异常。例如，捕获 SqlException 异常，可以导致抛出与业务相关的异常，例如，InvalidIsbnException 异常。

### 3. 重新抛出相同的异常

如果不应该改变异常的类型，就可以使用 throw 语句重新抛出相同的异常。使用 throw 但不传递异常对象，会抛出 catch 块的当前异常，并保存异常信息：

```
#line 2000
public static void HandleAndRethrow()
{
  try
  {
    ThrowAnException("test 3");
  }
  catch (Exception ex)
  {
    Console.WriteLine($"Log exception {ex.Message} and rethrow");
    throw; // line 2009
  }
}
```

有了这些代码，堆栈信息就不会丢失。异常最初是在 8002 行抛出，在第 2009 行重新抛出。行 114 包含调用 HandleAndRethrow 的委托 m：

```
Log exception test 3 and rethrow
test 3
at Program.ThrowAnException(String message) in Program.cs:line 8002
at Program.HandleAndRethrow() in Program.cs:line 2009
at Program.HandleAll() in Program.cs:line 114
```

### 4. 使用过滤器添加功能

使用 throw 语句重新抛出异常时，调用堆栈包含抛出的地址。使用异常过滤器，可以根本不改变调用堆栈。现在添加 when 关键字，传递过滤器方法。这个过滤器方法 Filter 记录消息，总是返回 false。这就是为什么 catch 块永远不会被调用的原因：

```
#line 1000
public void HandleWithFilter()
{
  try
  {
    ThrowAnException("test 4"); // line 1004
  }
  catch (Exception ex) when(Filter(ex))
  {
    Console.WriteLine("block never invoked");
  }
}
#line 1500
public bool Filter(Exception ex)
{
  Console.WriteLine($"just log {ex.Message}");
  return false;
}
```

现在看看堆栈跟踪，异常起源于 HandleAll 方法的第 114 行，它调用 HandleWithFilter，第 1004 行包含 ThrowAnException 的调用，第 8002 行抛出了异常：

```
just log test 4
test 4
at Program.ThrowAnException(String message) in Program.cs:line 8002
at Program.HandleWithFilter() in Program.cs:line 1004
at RethrowExceptions.Program.HandleAll() in Program.cs:line 114
```

**注意：**

异常过滤器的主要用法是基于值异常的过滤异常。异常过滤器也可以用于其他效果，比如写入日志信息，但不改变调用堆栈。然而，异常过滤器应该运行很快，所以应该只做简单的检查，避免副作用。日志记录是可执行异常的其中一个。

### 14.3.7　没有处理异常时发生的情况

有时抛出了一个异常后，代码中没有catch块能处理这类异常。前面的SimpleExceptions示例就说明了这种情况。例如，假定忽略FormatException异常和通用的catch块，则只有捕获IndexOutOfRangeException异常的块。此时，如果抛出一个FormatException异常，会发生什么情况呢？

答案是.NET 运行库会捕获它。本节后面将介绍如何嵌套 try 块——实际上在本示例中，就有一个在后台处理的嵌套的 try 块。.NET 运行库把整个程序放在另一个更大的 try 块中，对于每个.NET 程序它都会这么做。这个 try 块有一个 catch 处理程序，它可以捕获任何类型的异常。如果出现代码没有处理的异常，程序流就会退出程序，由.NET 运行库中的 catch 块捕获它。但是，事与愿违。代码的执行会立即终止，并给用户显示一个对话框，说明代码没有处理异常，并给出.NET 运行库能检索到的关于异常的详细信息。至少异常会被捕获。

一般情况下，如果编写一个可执行程序，就应捕获尽可能多的异常，并以合理的方式处理它们。如果编写一个库，最好捕获可以用有效方式处理的异常，或者在上下文中添加额外的信息，抛出其他异常类型，如上一节所示。假定调用代码可以处理它遇到的任何错误。

## 14.4　用户定义的异常类

上一节创建了一个用户定义的异常。下面介绍有关异常的第二个示例，这个示例称为 SolicitColdCall，它包含两个嵌套的 try 块，说明了如何定义自定义异常类，再从 try 块中抛出另一个异常。

这个示例假定一家销售公司希望有更多的客户。该公司的销售部门打算给一些人打电话，希望他们成为自己的客户。用销售行业的行话来讲，就是“陌生电话”(cold-calling)。为此，应有一个文本文件存储这些陌生人的姓名，该文件应有良好的格式，其中第一行包含文件中的人数，后面的行包含这些人的姓名。换言之，正确的格式如下所示。

```
4
George Washington
Benedict Arnold
John Adams
Thomas Jefferson
```

这个示例的目的是在屏幕上显示这些人的姓名(由销售人员读取)，这就是为什么只把姓名放在文件中，但没有电话号码的原因。

程序要求用户输入文件的名称，然后读取文件，并显示其中的人名。这听起来是一个很简单的任务，但也会出现两个错误，需要退出整个过程：

- 用户可能输入不存在的文件名。这作为 FileNotFound 异常来捕获。
- 文件的格式可能不正确，这里可能有两个问题。首先，文件的第一行不是整数。第二，文件中可能没有第一行指定的那么多人名。这两种情况都需要在一个自定义异常中处理，我们已经专门为此编写了 ColdCallFileFormatException 异常。

还会有其他问题，虽然不至于退出整个过程，但需要删除某个人名，继续处理文件中的下一个人名(因此这需要在内层的 try 块中处理)。一些人是商业间谍，为销售公司的竞争对手工作，显然，我们不希望不小心打电话给他们，让这些人知道我们要做的工作。为简单起见，假设姓名以 B 开头的那些人是商业间谍。这些人应在第一次准备数据文件时从文件中删除，但为防止有商业间谍混入，需要检查文件中的每个姓名，如果检测到一个商业间谍，就应抛出一个 SalesSpyFoundException 异常，当然，这是另一个自定义异常对象。

最后，编写一个类 ColdCallFileReader 来实现这个示例，该类维护与 cold-call 文件的连接，并从中检索数据。我们将以非常安全的方式编写这个类，如果其方法调用不正确，就会抛出异常。例如，如果在文件打开前，调用了读取文件的方法，就会抛出一个异常。为此，我们编写了另一个异常类 UnexpectedException。

### 14.4.1 捕获用户定义的异常

用户自定义异常的代码示例使用了如下名称空间：

System

System.IO

首先是 SolicitColdCall 示例的 Main()方法，它捕获用户定义的异常。注意，下面要调用 System.IO 名称空间和 System 名称空间中的文件处理类(代码文件 SolicitColdCall/Program.cs)。

```
public class Program
{
  public static void Main()
  {
    Console.Write("Please type in the name of the file " +
      "containing the names of the people to be cold called > ");
    string fileName = ReadLine();
    ColdCallFileReaderLoop1(fileName);
    Console.WriteLine();
    Console.ReadLine();
  }

  public static ColdCallfFileReaderLoop1(string filename)
  {
    var peopleToRing = new ColdCallFileReader();
    try
    {
      peopleToRing.Open(fileName);
      for (int i = 0; i < peopleToRing.NPeopleToRing; i++)
      {
        peopleToRing.ProcessNextPerson();
      }
      Console.WriteLine("All callers processed correctly");
    }
    catch(FileNotFoundException)
    {
      Console.WriteLine($"The file {fileName} does not exist");
    }
    catch(ColdCallFileFormatException ex)
    {
```

```
            Console.WriteLine($"The file {fileName} appears to have been corrupted");
            Console.WriteLine($"Details of problem are: {ex.Message}");
            if (ex.InnerException != null)
            {
              Console.WriteLine($"Inner exception was: {ex.InnerException.Message}");
            }
          }
          catch(Exception ex)
          {
            Console.WriteLine($"Exception occurred:\n{ex.Message}");
          }
          finally
          {
            peopleToRing.Dispose();
          }
        }
      }
```

这段代码基本上只是一个循环，用来处理文件中的人名。开始时，先让用户输入文件名，再实例化ColdCallFileReader类的一个对象，这个类稍后定义，正是这个类负责处理文件中数据的读取。注意，是在第一个try块的外部读取文件——这是因为这里实例化的变量需要在后面的catch 块和finally块中使用，如果在try块中声明它们，它们在try块的闭合花括号处就超出了作用域，这会导致异常。

在 try 块中打开文件(使用 ColdCallFileReader.Open()方法)，并循环处理其中的所有人名。ColdCallFileReader.ProcessNextPerson()方法会读取并显示文件中的下一个人名，而 ColdCallFileReader.NPeopleToRing 属性则说明文件中应有多少个人名(通过读取文件的第一行来获得)。有 3 个 catch 块，其中两个分别用于处理 FileNotFoundException 和 ColdCallFileFormatException 异常，第 3 个则用于处理任何其他.NET 异常。

在 FileNotFoundException 异常中，我们会为它显示一条消息，注意在这个 catch 块中，根本不会使用异常实例，原因是这个 catch 块用于说明应用程序的用户友好性。异常对象一般会包含技术信息，这些技术信息对开发人员很有用，但对于最终用户来说则没有什么用，所以本例将创建一条更简单的消息。

对于 ColdCallFileFormatException 异常的处理程序，则执行相反的操作，说明了如何获得更完整的技术信息，包括内层异常的细节(如果存在内层异常)。

最后，如果捕获到其他一般异常，就显示一条用户友好消息，而不是让这些异常由.NET 运行库处理。注意，我们选择不处理没有派生自 System.Exception 异常类的异常，因为不直接调用非.NET 的代码。

finally 块清理资源。在本例中，这是指关闭已打开的任何文件。ColdCallFileReader.Dispose()方法完成了这个任务。

**注意：**

*C#提供了一个 using 语句，编译器自己会在使用该语句的地方创建一个 try/finally 块，该块调用 finally 块中的 Dispose 方法。实现了一个 Dispose 方法的对象就可以使用 using 语句。第 17 章详细介绍了 using 语句。*

## 14.4.2　抛出用户定义的异常

下面看看处理文件读取，以及(可能)抛出用户定义的异常类 ColdCallFileReader 的定义。因为这个类维护一个外部文件连接，所以需要确保它根据第 4 章有关释放对象的规则，正确地释放它。这个类派生自 IDisposable 类。

首先声明一些私有字段(代码文件 SolicitColdCall/ColdCallFileReader.cs)：

```
public class ColdCallFileReader: IDisposable
{
  private FileStream _fs;
  private StreamReader _sr;
  private uint _nPeopleToRing;
  private bool _isDisposed = false;
  private bool _isOpen = false;
```

FileStream 和 StreamReader 都在 System.IO 名称空间中，它们都是用于读取文件的基类。FileStream 基类主要用于连接文件，StreamReader 基类则专门用于读取文本文件，并实现 Readline()方法，该方法读取文件中的一

行文本。第 22 章在深入讨论文件处理时将讨论 StreamReader 基类。

_isDisposed 字段表示是否调用了 Dispose()方法，我们选择实现 ColdCallFileReader 异常，这样，一旦调用了 Dispose()方法，就不能重新打开文件连接，重新使用对象了。isOpen 字段也用于错误检查——在本例中，检查 StreamReader 基类是否连接到打开的文件上。

打开文件和读取第一行的过程——告诉我们文件中有多少个人名——由 Open()方法处理：

```
public void Open(string fileName)
{
  if (_isDisposed)
  {
    throw new ObjectDisposedException("peopleToRing");
  }
  _fs = new FileStream(fileName, FileMode.Open);
  _sr = new StreamReader(_fs);
  try
  {
    string firstLine = _sr.ReadLine();
    _nPeopleToRing = uint.Parse(firstLine);
    _isOpen = true;
  }
  catch (FormatException ex)
  {
    throw new ColdCallFileFormatException(
      $"First line isn\'t an integer {ex}");
  }
}
```

与 ColdCallFileReader 异常类的所有其他方法一样，该方法首先检查在删除对象后，客户端代码是否不正确地调用了它，如果是，就抛出一个预定义的 ObjectDisposedException 异常对象。Open()方法也会检查_isDisposed 字段，看看是否已调用 Dispose()方法。因为调用 Dispose()方法会告诉调用者现在已经处理完对象，所以，如果已经调用了 Dispose()方法，就说明有一个试图打开新文件连接的错误。

接着，这个方法包含前两个内层的 try 块，其目的是捕获因为文件的第一行没有包含一个整数而抛出的任何错误。如果出现这个问题，.NET 运行库就抛出一个 FormatException 异常，该异常捕获并转换为一个更有意义的异常，这个更有意义的异常表示 cold-call 文件的格式有问题。注意 System.FormatException 异常表示与基本数据类型相关的格式问题，而不是与文件有关，所以在本例中它不是传递回主调例程的一个特别有用的异常。新抛出的异常会被最外层的 try 块捕获。因为这里不需要清理资源，所以不需要 finally 块。

如果一切正常，就把 isOpen 字段设置为 true，表示现在有一个有效的文件连接，可以从中读取数据。

ProcessNextPerson()方法也包含一个内层 try 块：

```
public void ProcessNextPerson()
{
  if (_isDisposed)
  {
    throw new ObjectDisposedException("peopleToRing");
  }
  if (!_isOpen)
  {
    throw new UnexpectedException(
      "Attempted to access coldcall file that is not open");
  }
  try
  {
    string name = _sr.ReadLine();
    if (name == null)
    {
      throw new ColdCallFileFormatException("Not enough names");
    }
    if (name[0] == 'B')
    {
      throw new SalesSpyFoundException(name);
    }
    Console.WriteLine(name);
  }
  catch(SalesSpyFoundException ex)
  {
    Console.WriteLine(ex.Message);
  }
```

```
    finally
    {
    }
  }
```

这里可能存在两个与文件相关的错误(假定实际上有一个打开的文件连接，ProcessNextPerson()方法会先进行检查)。第一，读取下一个人名时，可能发现这是一个商业间谍。如果发生这种情况，在这个方法中就使用第一个 catch 块捕获异常。因为这个异常已经在循环中被捕获，所以程序流会继续在程序的 Main()方法中执行，处理文件中的下一个人名。

如果读取下一个人名，发现已经到达文件的末尾，就会发生错误。StreamReader 对象的 ReadLine()方法的工作方式是：如果到达文件末尾，它就会返回一个 null，而不是抛出一个异常。所以，如果找到一个 null 字符串，就说明文件的格式不正确，因为文件的第一行中的数字要比文件中的实际人数多。如果发生这种错误，就抛出一个 ColdCallFileFormatException 异常，它由外层的异常处理程序捕获(使程序终止执行)。

同样，这里不需要 finally 块，因为没有要清理的资源，但这次要放置一个空的 finally 块，表示在这里可以完成用户希望完成的任务。

这个示例就要完成了。ColdCallFileReader 异常类还有另外两个成员：NPeopleToRing 属性返回文件中应有的人数，Dispose()方法可以关闭已打开的文件。注意 Dispose()方法仅返回它是否被调用——这是实现该方法的推荐方式。它还检查在关闭前是否有一个文件流要关闭。这个例子说明了防御编码技术：

```
public uint NPeopleToRing
{
  get
  {
    if (_isDisposed)
    {
      throw new ObjectDisposedException("peopleToRing");
    }
    if (!_isOpen)
    {
      throw new UnexpectedException(
        "Attempted to access cold-call file that is not open");
    }
    return _nPeopleToRing;
  }
}

public void Dispose()
{
  if (_isDisposed)
  {
    return;
  }
  _isDisposed = true;
  _isOpen = false;
  _fs?.Dispose();
  _fs = null;
}
```

### 14.4.3　定义用户定义的异常类

最后，需要定义 3 个异常类。定义自己的异常非常简单，因为几乎不需要添加任何额外的方法。只需要实现构造函数，确保基类的构造函数正确调用即可。下面是实现 SalesSpyFoundException 异常类的完整代码(代码文件 SolicitColdCall/SalesSpyFoundException.cs)：

```
public class SalesSpyFoundException: Exception
{
  public SalesSpyFoundException(string spyName)
    : base($"Sales spy found, with name {spyName}")
  {
  }

  public SalesSpyFoundException(string spyName, Exception innerException)
    : base($"Sales spy found with name {spyName}", innerException)
  {
  }
}
```

注意，这个类派生自 Exception 异常类，正是我们期望的自定义异常。实际上，如果要更正式地创建它，可以把它放在一个中间类中，例如，ColdCallFileException 异常类，让它派生于 Exception 异常类，再从这个类派生出两个异常类，并确保处理代码可以很好地控制哪个异常处理程序处理哪个异常即可。但为了使这个示例比较简单，就不这么操作了。

在 SalesSpyFoundException 异常类中，处理的内容要多一些。假定传送给它的构造函数的信息仅是找到的间谍名，从而把这个字符串转换为含义更明确的错误信息。我们还提供了两个构造函数，其中一个构造函数的参数只是一条消息，另一个构造函数的参数是一个内层异常。在定义自己的异常类时，至少把这两个构造函数都包括进来(尽管以后将不能在示例中使用 SalesSpyFoundException 异常类的第 2 个构造函数)。

对于 ColdCallFileFormatException 异常类，规则是一样的，但不必对消息进行任何处理(代码文件 SolicitColdCall/ColdCallFileFormatException.cs)：

```
public class ColdCallFileFormatException: Exception
{
  public ColdCallFileFormatException(string message)
    : base(message)
  {
  }

  public ColdCallFileFormatException(string message, Exception innerException)
    : base(message, innerException)
  {
  }
}
```

最后是 UnexpectedException 异常类，它看起来与 ColdCallFileFormatException 异常类是一样的(代码文件 SolicitColdCall/UnexpectedException.cs)：

```
public class UnexpectedException: Exception
{
  public UnexpectedException(string message)
    : base(message)
  {
  }

  public UnexpectedException(string message, Exception innerException)
    : base(message, innerException)
  {
  }
}
```

下面准备测试该程序。首先，使用 people.txt 文件，其内容已经在前面列出了。

```
4
George Washington
Benedict Arnold
John Adams
Thomas Jefferson
```

它有 4 个名字(与文件中第一行给出的数字匹配)，包括一个间谍。接着，使用下面的 people2.txt 文件，它有一个明显的格式错误：

```
49
George Washington
Benedict Arnold
John Adams
Thomas Jefferson
```

最后，尝试执行该例子，但指定一个不存在的文件名 people3.txt，对这 3 个文件名运行程序 3 次，得到的结果如下：

```
SolicitColdCall
Please type in the name of the file containing the names of the people to be cold
called > people.txt
George Washington
Sales spy found, with name Benedict Arnold
John Adams
Thomas Jefferson
All callers processed correctly
SolicitColdCall
```

```
Please type in the name of the file containing the names of the people to be cold
called > people2.txt
George Washington
Sales spy found, with name Benedict Arnold
John Adams
Thomas Jefferson
The file people2.txt appears to have been corrupted.
Details of the problem are: Not enough names
SolicitColdCall
Please type in the name of the file containing the names of the people to be cold
called > people3.txt
The file people3.txt does not exist.
```

最后，这个应用程序说明了处理程序中可能存在的错误和异常的许多不同方式。

## 14.5　调用者信息

在处理错误时，获得错误发生位置的信息常常是有帮助的。本章全面介绍的#line 预处理器指令用于改变代码的行号，获得调用堆栈的更好信息。为了从代码中获得行号、文件名和成员名，可以使用 C# 编译器直接支持的特性和可选参数。这些特性包括 CallerLineNumber、CallerFilePath 和 CallerMemberName，它们定义在 System.Runtime.CompilerServices 名称空间中，可以应用到参数上。对于可选参数，当没有提供调用信息时，编译器会在调用方法时为它们使用默认值。有了调用者信息特性，编译器不会填入默认值，而是填入行号、文件路径和成员名称。

代码示例 CallerInformation 使用如下名称空间：

System

System.Runtime.CompilerServices

下面代码段中的 Log 方法演示了这些特性的用法。这段代码将信息写入控制台中(代码文件 CallerInformation/Program.cs)：

```
public void Log([CallerLineNumber] int line = -1,
[CallerFilePath] string path = null,
[CallerMemberName] string name = null)
{
  Console.WriteLine((line < 0) ? "No line" : "Line " + line);
  Console.WriteLine((path == null) ? "No file path" : path);
  Console.WriteLine((name == null) ? "No member name" : name);
  Console.WriteLine();
}
```

下面在几种不同的场景中调用该方法。在下面的Main()方法中，分别使用Program类的一个实例来调用Log()方法，在属性的 set 访问器中调用 Log()方法，以及在一个 lambda 表达式中调用 Log()方法。这里没有为该方法提供参数值，所以编译器会为其填入值：

```
public static void Main()
{
  var p = new Program();
  p.Log();
  p.SomeProperty = 33;
  Action a1 = () => p.Log();
  a1();
}

private int _someProperty;
public int SomeProperty
{
  get => _someProperty;
  set
  {
    Log();
    _someProperty = value;
  }
}
```

运行此程序的结果如下所示。在调用 Log()方法的地方，可以看到行号、文件名和调用者的成员名。对于 Main()方法中调用的 Log()方法，成员名为 Main。对于属性 SomeProperty 的 set 访问器中调用的 Log()方法，成

员名为 SomeProperty。lambda 表达式中的 Log()方法没有显示生成的方法名，而是显示了调用该 lambda 表达式的方法的名称(Main)，这当然更加有用。

```
Line 12
c:\ProCSharp\ErrorsAndExceptions\CallerInformation\Program.cs
Main
Line 26
c:\ProCSharp\ErrorsAndExceptions\CallerInformation\Program.cs
SomeProperty
Line 14
c:\ProCSharp\ErrorsAndExceptions\CallerInformation\Program.cs
Main
```

在构造函数中使用 Log()方法时，调用者成员名显示为 ctor。在析构函数中，调用者成员名为 Finalize，因为它是生成的方法的名称。

**注意：**
析构函数和终结器参见第 17 章。

**注意：**
CallerMemberName 属性的一个很好的用途是用在 INotifyPropertyChanged 接口的实现中。该接口要求在方法的实现中传递属性的名称。在本书的几个章节中都可以看到这个接口的实现，例如第 34 章。

# 14.6 小结

本章介绍了 C#通过异常处理错误情况的多种机制，我们不仅可以输出代码中的一般错误代码，还可以用指定的方式处理最特殊的错误情况。有时一些错误情况是通过.NET Framework 本身提供的，有时则需要编写自己的错误情况，如本章的例子所示。在这两种情况下，都可以采用许多方式来保护应用程序的工作流，使之不出现不必要和危险的错误。

第 15 章将学习异步编程的重要关键字 async 和 await。

# 第 15 章

# 异 步 编 程

**本章要点**

- 异步编程的重要性
- 异步模式
- 异步编程的基础
- 异步方法的错误处理
- Windows 应用程序的异步编程

**本章源代码下载地址(wrox.com)：**

打开 www.wrox.com 的 Download Code 选项卡可下载本章源代码。源代码也可以在 Async 目录的 https://github.com/ProfessionalCSharp/ProfessionalCSharp7 中找到。本章代码分为以下几个主要的示例文件：

- AsyncHistory
- Foundations
- Error Handling
- AsyncWindowsApp

## 15.1 异步编程的重要性

.NET Framework 4.5 将任务并行库(Task Para uel Library，TPL)添加到.NET 中，以使并行编程更容易。C# 5.0 增加了两个关键字来简化异步编程：async 和 await。这两个关键字将是本章的重点。

使用异步编程，方法调用是在后台运行(通常在线程或任务的帮助下)，并且不会阻塞调用线程。

本章将学习 3 种不同模式的异步编程：异步模式、基于事件的异步模式和基于任务的异步模式(Task-based Asynchronous Pattern，TAP)。TAP 是利用 async 和 await 关键字来实现的。通过这里的比较，将认识到异步编程新模式的真正优势。

讨论过不同的模式之后，通过创建任务和使用异步方法，来介绍异步编程的基础知识。还会论述延续任务和同步上下文的相关内容。

与异步任务一样，错误处理也需要特别重视。有些错误要采用不同的处理方式。

本章的最后一部分讨论了通用 Windows 应用程序的特定场景，学习异步编程所需要了解的内容。

**注意：**
第 21 章介绍了并行编程的相关内容。

如果应用程序没有立刻响应用户的请求，会让用户反感。用鼠标操作，我们习惯了出现延迟，过去几十年都是这样操作的。有了触摸 UI，应用程序要求立刻响应用户的请求。否则，用户就会不断重复同一个动作。

因为在旧版本的.NET Framework 中用异步编程非常不方便，所以并没有总是这样做。Visual Studio 旧版本是经常阻塞 UI 线程的应用程序之一。例如，在 Visual Studio 的旧版本中，打开一个包含数百个项目的解决方案，这意味可能需要等待很长的时间。Visual Studio 2017 提供了轻量级的解决方案加载(Lightweight Solution Load)特性，它只在需要时加载项目，且先加载选定的项目。从 Visual Studio 2015 开始，NuGet 包管理器不再实现为模式对话框。新的 NuGet 包管理器可以异步加载包的信息，同时做其他工作。这是异步编程内置到 Visual Studio 2015 中带来的重要变化之一。

很多.NET Framework 的 API 都提供了同步版本和异步版本。因为同步版本的 API 用起来更为简单，所以常常在不适合使用时也用了同步版本的 API。在新的 Windows Runtime(WinRT)中，如果一个 API 调用时间超过 40ms，就只能使用其异步版本。自从 C# 5.0 开始，异步编程和同步编程一样简单，所以用异步 API 应该不会有任何的障碍。

## 15.2 异步编程的.NET 历史

在学习新的 async 和 await 关键字之前，先看看.NET Framework 的异步模式。从.NET Framework 1.0 开始就提供了异步特性，而且.NET Framework 的许多类都实现了一个或者多个异步模式。

下面开始执行同步网络调用，然后介绍不同的异步模式：

- 异步模式
- 基于事件的异步模式
- 基于任务的异步模式

异步模式是处理异步特性的第一种方式，它不仅可以使用几个 API，还可以使用基本功能(如委托类型)。

因为在 Windows Forms 和 WPF 中，用异步模式更新界面非常复杂，所以.NET Framework 2.0 推出了基于事件的异步模式。在这种模式中，事件处理程序是被拥有同步上下文的线程调用，所以更新界面很容易用这种模式处理。在此之前，这种模式也称为异步组件模式。

在.NET Framework 4.5 中，推出了另外一种方式来实现异步编程：基于任务的异步模式(TAP)。这种模式是基于 Task 类型，并通过 async 和 await 关键字使用编译器功能。

HistorySample 的示例代码至少使用 C# 7.1 和以下名称空间：

System
System.IO
System.Net
System.Threading.Tasks

发出 HTTP 请求的示例应用程序是 System.Net API 提供同步和异步 API 中的一个好例子。

### 15.2.1 同步调用

下面从使用 WebClient 类的同步版本开始。这个类提供了几个同步的 API，如 DownloadString、DownloadFile 和 DownloadData。在下面的代码片段中，DownloadString 发出一个 HTTP 请求并在字符串内容中写入响应。该字符串的子字符串被写入控制台(代码文件 AsyncHistory/Program.cs)：

```
private const string url = "http://www.cninnovation.com";

private static void SynchronizedAPI()
```

```
{
  Console.WriteLine(nameof(SynchronizedAPI));
  using (var client = new WebClient())
  {
    string content = client.DownloadString(url);
    Console.WriteLine(content.Substring(0, 100));
  }
  Console.WriteLine();
}
```

方法 DownloadString 阻塞调用线程，直到返回结果。从客户端应用程序的用户界面线程中调用这个方法并不好，因为它会阻塞用户界面。等待对用户来说是不愉快的，因为在这个网络调用中应用程序没有响应。

### 15.2.2　异步模式

异步调用的一种方式是使用异步模式。.NET 的许多 API 都提供异步模式。对于.NET Framework，委托类型也支持这种模式。请注意，使用.NET Core 调用委托的这些方法时，会抛出一个异常，其中包含平台不支持的信息。

异步模式定义了 BeginXXX 方法和 EndXXX 方法。例如，如果有一个同步方法 DownloadString，其异步版本就是 BeginDownloadString 和 EndDownloadString 方法。BeginXXX 方法接受其同步方法的所有输入参数，EndXXX 方法使用同步方法的所有输出参数，并按照同步方法的返回类型来返回结果。使用异步模式时，BeginXXX 方法还定义了一个 AsyncCallback 参数，用于接受在异步方法执行完成后调用的委托。BeginXXX 方法返回 IAsyncResult，用于验证调用是否已经完成，并且一直等到方法的执行结束。

WebClient 类没有提供异步模式的实现方式，但是可以用 WebRequest 类来替代，因为该类通过 BeginGetResponse 和 EndGetResponse 方法提供这种模式(GetResponse 是这个 API 的同步版本)。

在下面的代码片段中，使用WebRequest类的Create方法创建WebRequest。使用这个请求对象，BeginGetResponse方法将异步HTTP GET请求发到服务器。调用线程没有被阻塞。该方法的第一个参数是AsyncCallback。这是一个通过IAsyncResult参数引用void方法的委托。实现代码是使用本地函数ReadResponse完成的。一旦网络请求完成，就会调用该方法。在实现代码中，再次通过request对象使用GetResponseStream检索结果。在代码示例中，Stream和StreamReader用于访问返回的字符串内容(代码文件AsyncHistory/Program.cs)：

```
private static void AsynchronousPattern()
{
  Console.WriteLine(nameof(AsynchronousPattern));
  WebRequest request = WebRequest.Create(url);
  IAsyncResult result = request.BeginGetResponse(ReadResponse, null);

  void ReadResponse(IAsyncResult ar)
  {
    using (WebResponse response = request.EndGetResponse(ar))
    {
      Stream stream = response.GetResponseStream();
      var reader = new StreamReader(stream);
      string content = reader.ReadToEnd();
      Console.WriteLine(content.Substring(0, 100));
      Console.WriteLine();
    }
  }
}
```

由于在实现代码中使用了本地函数，因此从外部作用域的请求变量可以通过本地函数的闭包功能直接访问。lambda 表达式也具有类似的行为。如果要使用单独的方法，则必须将请求对象传递给此方法。为此需要将请求对象传递为 BeginGetResponse 方法的第二个参数。可以使用 IAsyncResult 的 AsyncState 属性在被调用的方法中检索这个参数。

> **注意：**
> 本地函数参见第 13 章。

在 UI 应用程序中使用异步模式有一个问题：从 AsyncCallback 调用的方法没有在 UI 线程中运行，因此如

果不切换到 UI 线程，就不能访问 UI 元素的成员，而是抛出一个异常 The calling thread cannot access this object because a different thread owns it.(调用线程不能访问这个对象，因为另一个线程拥有它)。为了简化这一过程，.NET Framework 2.0 引入了基于事件的异步模式，这更易于处理 UI 更新。接下来将讨论此模式。

### 15.2.3 基于事件的异步模式

EventBasedAsyncPattern 方法使用了基于事件的异步模式。这个模式定义了一个带有“Async”后缀的方法。示例代码再次使用了 WebClient 类。对于同步方法 DownloadString，WebClient 类提供了一个异步变体方法 DownloadStringAsync。当请求完成时，会触发 DownloadStringCompleted 事件。使用此事件的事件处理程序，可以检索结果。DownloadStringCompleted 事件类型为 DownloadStringCompletedEventHandler。第二个参数是 DownloadStringCompletedEventArgs 类型。这个参数通过 Result 属性返回结果字符串(代码文件 AsyncHistory/Program.cs)：

```
private static void EventBasedAsyncPattern()
{
  Console.WriteLine(nameof(EventBasedAsyncPattern));
  using (var client = new WebClient())
  {
    client.DownloadStringCompleted += (sender, e) =>
    {
      Console.WriteLine(e.Result.Substring(0, 100));
    };
    client.DownloadStringAsync(new Uri(url));
    Console.WriteLine();
  }
}
```

使用 DownloadStringCompleted 事件，事件处理程序将通过保存同步上下文的线程来调用。在 Windows 窗体、WPF 和 UWP 中，这就是 UI 线程。因此，可以直接从事件处理程序中访问 UI 元素。与异步模式相比，这是该模式的一大优点。

基于事件的异步模式和同步编程之间的区别在于方法调用的顺序；与同步方法调用相比，顺序颠倒了。调用异步方法之前，需要定义这个方法完成时发生什么。15.2.4 小节将进入异步编程的新世界：利用 async 和 await 关键字。

### 15.2.4 基于任务的异步模式

在.NET Framework 4.5 中，更新了 WebClient 类，还提供了基于任务的异步模式(TAP)。该模式定义了一个带有“Async”后缀的方法，并返回一个 Task 类型。由于 WebClient 类已经提供了一个带 Async 后缀的方法来实现基于任务的异步模式，因此新方法名为 DownloadStringTaskAsync。

DownloadStringTaskAsync 方法声明为返回 Task<string>。但是，不需要声明一个 Task<string>类型的变量来设置 DownloadStringTaskAsync 方法返回的结果。只要声明一个 String 类型的变量，并使用 await 关键字。await 关键字会解除线程(这里是 UI 线程)的阻塞，完成其他任务。当 DownloadStringTaskAsync 方法完成其后台处理后，UI 线程就可以继续，从后台任务中获得结果，赋值给字符串变量 resp。然后执行 await 关键字后面的代码(代码文件 AsyncHistory/Program.cs)：

```
private static async Task TaskBasedAsyncPatternAsync()
{
  Console.WriteLine(nameof(TaskBasedAsyncPatternAsync));
  using (var client = new WebClient())
  {
    string content = await client.DownloadStringTaskAsync(url);
    Console.WriteLine(content.Substring(0, 100));
    Console.WriteLine();
  }
}
```

**注意：**
async 关键字创建了一个状态机，类似于 yield return 语句。参见第 7 章。

现在，代码就简单多了。没有阻塞，也不需要切换回 UI 线程，这些都是自动实现的。代码顺序也和惯用的同步编程一样。

**注意：**
更现代的 HTTP 客户端是用类 HttpClient 实现的。这个类提供的异步方法支持基于任务的异步模式。如何使用这个类在第 23 章中解释。

### 15.2.5　异步 Main()方法

控制台应用程序的入口点是 Main ()方法，它应用 async 修饰符允许在实现中使用 await 关键字。使用 Main()方法的这个声明返回一个任务需要 C# 7.1(代码文件 AsyncHistory/Program.cs)：

```
static async Task Main()
{
  SynchronizedAPI();
  AsynchronousPattern();
  EventBasedAsyncPattern();
  await TaskBasedAsyncPatternAsync();
  Console.ReadLine();
}
```

**注意：**
要指定 C#编译器的 7.1 版本，需要将 LangVersion 元素添加到 csproj 项目文件中，或者可以在 *Advanced Build Project Settings* 中对 Visual Studio 进行更改。

认识到 async 和 await 关键字的优势后，15.3 节将讨论异步编程的基础。

## 15.3　异步编程的基础

async 和 await 关键字只是编译器功能。编译器会用 Task 类创建代码。如果不使用这两个关键字，也可以用 C# 4.0 和 Task 类的方法来实现同样的功能，只是没有那么方便。

本节介绍了编译器用 async 和 await 关键字能做什么，如何采用简单的方式创建异步方法，如何并行调用多个异步方法，以及如何修改已经实现异步模式的类，以使用新的关键字。

所有 Foundations 的示例代码都使用了如下名称空间：

System
System.Collections.Generic
System.IO
System.Linq
System.Net
System.Runtime.CompilerServices
System.Threading
System.Threading.Tasks

**注意：**
这个可下载的示例应用程序使用了命令行参数，因此可以轻松地验证每个场景。例如，使用 dotnet CLI，可以通过命令：dotnet run ---async 传递-async 命令行参数。使用 Visual Studio，还可以在 *Debug Project Settings* 中配置应用程序的参数。

为了更好地理解发生了什么，创建TraceThreadAndTask方法，将线程和任务信息写入控制台。Task.CurrentId返回任务的标识符。Thread.CurrentThread.ManagedThreadId返回当前线程的标识符(代码文件Foundations/Program.cs)：

```
public static void TraceThreadAndTask(string info)
{
  string taskInfo = Task.CurrentId == null ? "no task" : "task " +
    Task.CurrentId;

  Console.WriteLine($"{info} in thread {Thread.CurrentThread.ManagedThreadId}" +
    $"and {taskInfo}");
}
```

## 15.3.1　创建任务

下面从同步方法 Greeting 开始，该方法等待一段时间后，返回一个字符串(代码文件 Foundations/Program.cs)：

```
static string Greeting(string name)
{
  TraceThreadAndTask($"running {nameof(Greeting)}");
  Task.Delay(3000).Wait();
  return $"Hello, {name}";
}
```

定义方法 GreetingAsync，可以使方法异步化。基于任务的异步模式指定，在异步方法名后加上 Async 后缀，并返回一个任务。异步方法 GreetingAsync 和同步方法 Greeting 具有相同的输入参数，但是它返回的是 Task<string>。Task<string>定义了一个返回字符串的任务。一个比较简单的做法是用 Task.Run 方法返回一个任务。泛型版本的 Task.Run<string>()创建一个返回字符串的任务。由于编译器已经知道实现的返回类型(Greeting 返回字符串)，因此还可以使用 Task.Run()来简化实现代码：

```
static Task<string> GreetingAsync(string name) =>
  Task.Run<string>(() =>
  {
    TraceThreadAndTask($"running {nameof(GreetingAsync)}");
    return Greeting(name);
  });
```

## 15.3.2　调用异步方法

可以使用 await 关键字来调用返回任务的异步方法 GreetingAsync。使用 await 关键字需要有用 async 修饰符声明的方法。在 GreetingAsync 方法完成前，该方法内的其他代码不会继续执行。但是，启动 CallerWithAsync 方法的线程可以被重用。该线程没有被阻塞(代码文件 Foundations/Program.cs)：

```
private async static void CallerWithAsync()
{
  TraceThreadAndTask($"started {nameof(CallerWithAsync)}");
  string result = await GreetingAsync("Stephanie");
  Console.WriteLine(result);
  TraceThreadAndTask($"ended {nameof(CallerWithAsync)}");
}
```

运行应用程序时，可以从第一个输出中看到没有任务。GreetingAsync 方法在一个任务中运行，这个任务使用的线程与调用者不同。然后，同步 Greeting 方法在此任务中运行。当 Greeting 方法返回时，GreetingAsync 方法返回，在等待之后，作用域返回到 CallerWithAsync 方法中。现在，CallerWithAsync 方法在不同的线程中运行。不再有任务了，但是尽管这个方法是从线程 2 开始的，但是在使用了 await 线程 3 之后。await 确保在任务完成后继续执行，但现在它使用了另一个线程。这种行为在控制台应用程序和具有同步上下文的应用程序之间是不同的：

```
started CallerWithAsync in thread 2 and no task
running GreetingAsync in thread 3 and task 1
running Greeting in thread 3 and task 1
Hello, Stephanie
ended CallerWithAsync in thread 3 and no task
```

如果异步方法的结果不传递给变量，也可以直接在参数中使用 await 关键字。在这里，GreetingAsync 方法

返回的结果将像前面的代码片段一样等待，但是这一次的结果会直接传给 Console.WriteLine 方法：

```
private async static void CallerWithAsync2()
{
  TraceThreadAndTask($"started {nameof(CallerWithAsync2)}");
  Console.WriteLine(await GreetingAsync("Stephanie"));
  TraceThreadAndTask($"ended {nameof(CallerWithAsync2)}");
}
```

**注意：**

在 C# 7 中，async 修饰符可以用于返回 void 的方法，或者用于返回一个提供 GetAwaiter 方法的对象。.NET 提供 Task 和 ValueTask 类型。通过 Windows 运行库，还可以使用 IAsyncOperation。应该避免给带有 void 的方法使用 async 修饰符。

15.3.3 小节会介绍是什么驱动了 await 关键字，在后台使用了延续任务。

### 15.3.3　使用 Awaiter

可以对任何提供 GetAwaiter 方法并返回 awaiter 的对象使用 async 关键字。awaiter 用 OnCompleted 方法实现 INotifyCompletion 接口。此方法在任务完成时调用。下面的代码片段不是在任务中使用 await，而是使用任务的 GetAwaiter 方法。Task 类的 GetAwaiter 返回一个 TaskAwaiter。使用 OnCompleted 方法，分配一个在任务完成时调用的本地函数(代码文件 Foundations/Program.cs)：

```
private static void CallerWithAwaiter()
{
  TraceThreadAndTask($"starting {nameof(CallerWithAwaiter)}");
  TaskAwaiter<string> awaiter = GreetingAsync("Matthias").GetAwaiter();
  awaiter.OnCompleted(OnCompleteAwaiter);

  void OnCompleteAwaiter()
  {
    Console.WriteLine(awaiter.GetResult());
    TraceThreadAndTask($"ended {nameof(CallerWithAwaiter)}");
  }
}
```

运行应用程序时，结果类似于你使用 wait 关键字的情形：

```
starting CallerWithAwaiter in thread 2 and no task
running GreetingAsync in thread 3 and task 1
running Greeting in thread 3 and task 1
Hello, Matthias
ended CallerWithAwaiter in thread 3 and no task
```

编译器把 await 关键字后的所有代码放进 OnCompleted 方法的代码块中来转换 await 关键字。

### 15.3.4　延续任务

还可以使用 Task 对象的特性来处理任务的延续。GreetingAsync 方法返回一个 Task<string>对象。该 Task<string>对象包含任务创建的信息，并保存到任务完成。Task 类的 ContinueWith 方法定义了任务完成后就调用的代码。指派给 ContinueWith 方法的委托接收将已完成的任务作为参数传入，使用 Result 属性可以访问任务返回的结果(代码文件 Foundations/Program.cs)：

```
private static void CallerWithContinuationTask()
{
  TraceThreadAndTask("started CallerWithContinuationTask");

  var t1 = GreetingAsync("Stephanie");

  t1.ContinueWith(t =>
  {
    string result = t.Result;
    Console.WriteLine(result);

    TraceThreadAndTask("ended CallerWithContinuationTask");
  });
```

```
}
```

### 15.3.5 同步上下文

如果验证方法中使用的线程，会发现 CallerWithAsync 方法、CallerWithAwaiter 方法和 CallerWithContinuationTask 方法，在方法的不同生命阶段使用了不同的线程。一个线程用于调用 GreetingAsync 方法，另外一个线程执行 await 关键字后面的代码，或者继续执行 ContinueWith 方法内的代码块。

使用一个控制台应用程序，通常不会有什么问题。但是，必须保证在所有应该完成的后台任务完成之前，至少有一个前台线程仍然在运行。示例应用程序调用 Console.ReadLine 来保证主线程一直在运行，直到按下返回键。

为了执行某些动作，有些应用程序会绑定到指定的线程上(例如，在 WPF 或 Windows 应用程序中，只有 UI 线程才能访问 UI 元素)，这将会是一个问题。

如果使用async和await关键字，当await完成之后，不需要进行任何特别处理，就能访问UI线程。默认情况下，生成的代码会把线程转换到拥有同步上下文的线程中。WPF应用程序设置了DispatcherSynchronizationContext 属性，Windows Forms应用程序设置了WindowsFormsSynchronizationContext属性。Windows应用程序使用 WinRTSynchronizationContext。如果调用异步方法的线程分配给了同步上下文，await完成之后将继续执行。默认情况下，使用了同步上下文。如果不使用相同的同步上下文，则必须调用Task方法ConfigureAwait (continueOnCapturedContext：false)。例如，一个Windows应用程序，其await后面的代码没有用到任何的UI元素。在这种情况下，避免切换到同步上下文会执行得更快。

### 15.3.6 使用多个异步方法

在一个异步方法中，可以调用一个或多个异步方法。如何编写代码，取决于一个异步方法的结果是否依赖于另一个异步方法。

#### 1. 按顺序调用异步方法

使用 await 关键字可以调用每个异步方法。在有些情况下，如果一个异步方法依赖另一个异步方法的结果，await 关键字就非常有用。在这里，GreetingAsync 异步方法的第二次调用完全独立于其第一次调用的结果。这样，如果每个异步方法都不使用 await，那么整个 MultipleAsyncMethods 异步方法将更快地返回结果，如下所示(代码文件 Foundations/Program.cs)：

```
private async static void MultipleAsyncMethods()
{
  string s1 = await GreetingAsync("Stephanie");
  string s2 = await GreetingAsync("Matthias");
  Console.WriteLine($"Finished both methods.{Environment.NewLine} " +
    $"Result 1: {s1}{Environment.NewLine} Result 2: {s2}");
}
```

#### 2. 使用组合器

如果异步方法不依赖于其他异步方法，则每个异步方法都不使用 await，而是把每个异步方法的返回结果赋值给 Task 变量，就会运行得更快。GreetingAsync 方法返回 Task<string>。这些方法现在可以并行运行了。组合器可以帮助实现这一点。一个组合器可以接受多个同一类型的参数，并返回同一类型的值。多个同一类型的参数被组合成一个参数来传递。Task 组合器接受多个 Task 对象作为参数，并返回一个 Task。

示例代码调用 Task.WhenAll 组合器方法，它可以等待，直到两个任务都完成(代码文件 Foundations/Program.cs)。

```
private async static void MultipleAsyncMethodsWithCombinators1()
{
  Task<string> t1 = GreetingAsync("Stephanie");
  Task<string> t2 = GreetingAsync("Matthias");
  await Task.WhenAll(t1, t2);
  Console.WriteLine($"Finished both methods.{Environment.NewLine} " +
    $"Result 1: {t1.Result}{Environment.NewLine} Result 2: {t2.Result}");
}
```

Task 类定义了 WhenAll 和 WhenAny 组合器。从 WhenAll 方法返回的 Task，是在所有传入方法的任务都完成了才会返回 Task。从 WhenAny 方法返回的 Task，是在其中一个传入方法的任务完成了就会返回 Task。

Task 类型的 WhenAll 方法定义了几个重载版本。如果所有的任务返回相同的类型，那么该类型的数组可用于 await 返回的结果。GreetingAsync 方法返回一个 Task<string>，等待返回的结果是一个字符串(string)形式。因此，Task.WhenAll 可用于返回一个字符串数组：

```
private async static void MultipleAsyncMethodsWithCombinators2()
{
  Task<string> t1 = GreetingAsync("Stephanie");
  Task<string> t2 = GreetingAsync("Matthias");
  string[] result = await Task.WhenAll(t1, t2);
  Console.WriteLine($"Finished both methods.{Environment.NewLine} " +
    $"Result 1: {result[0]}{Enviornment.NewLine} Result 2: {result[1]}");
}
```

只有等待的所有任务都完成时某个任务才能继续，WhenAll 方法就有实际用途。当调用任务在等待完成的任何任务都完成才能执行操作时，可以使用 WhenAny 方法。它可以使用任务的结果继续。

### 15.3.7　使用 ValueTasks

C# 7 带有更灵活的 await 关键字；它现在可以等待任何提供 GetAwaiter 方法的对象。一种可用于等待的新类型是 ValueTask。与 Task 类相反，ValueTask 是一个结构。这具有性能优势，因为 ValueTask 在堆上没有对象。

与异步方法调用相比，Task 对象的实际开销是多少？需要异步调用的方法通常比堆上的对象有更多的开销。大多数时候，堆上 Task 对象的开销是可以忽略的，但并不总是这样。例如，某方法可以有一个路径，其中数据是从一个具有异步 API 的服务中检索出来的。通过这种数据检索，数据就写入到本地缓存中。第二次调用该方法时，可以以快速的方式检索数据，而不需要创建 Task 对象。

示例方法 GreetingValueTaskAsync 正是这样做的。如果该名称已存在于字典中，则结果返回为 ValueTask。如果名称不在字典中，将调用 GreetingAsync 方法，该方法返回一个 Task。在此任务中等待检索结果时，将再次返回 ValueTask(代码文件 Foundations/Program.cs)：

```
private readonly static Dictionary<string, string> names = new Dictionary<string, string>();

static async ValueTask<string> GreetingValueTaskAsync(string name)
{
  if (names.TryGetValue(name, out string result))
  {
    return result;
  }
  else
  {
    result = await GreetingAsync(name);
    names.Add(name, result);
    return result;
  }
}
```

UseValueTask 方法使用相同的名称调用 GreetingValueTaskAsync 方法两次。第一次使用 GreetingAsync 方法检索数据；第二次，数据在字典中找到并从那里返回：

```
private static async void UseValueTask()
{
  string result = await GreetingValueTaskAsync("Katharina");
  Console.WriteLine(result);
  string result2 = await GreetingValueTaskAsync("Katharina");
  Console.WriteLine(result2);
}
```

如果方法不使用 async 修饰符，而需要返回 ValueTask，就可以使用传递结果或者传递 Task 对象的构造函数创建 ValueTask 对象：

```
static ValueTask<string> GreetingValueTask2Async(string name)
{
  if (names.TryGetValue(name, out string result))
  {
```

```
    return new ValueTask<string>(result);
  }
  else
  {
    Task<string> t1 =  GreetingAsync(name);

    TaskAwaiter<string> awaiter = t1.GetAwaiter();
    awaiter.OnCompleted(OnCompletion);
    return new ValueTask<string>(t1);

    void OnCompletion()
    {
      names.Add(name, awaiter.GetResult());
    }
  }
}
```

### 15.3.8 转换异步模式

并非.NET Framework 的所有类都引入了新的异步方法。在使用框架中的不同类时会发现，还有许多类只提供了 BeginXXX 方法和 EndXXX 方法的异步模式，没有提供基于任务的异步模式。但是，可以把异步模式转换为基于任务的异步模式。

这个示例使用 HttpWebRequest 类和 BeginGetResponse 方法将该方法转换为基于任务的异步模式。Task.Factory.FromAsync 是一个泛型方法，它提供了一些重载版本，将异步模式转换为基于任务的异步模式。对于示例应用程序，当调用 HttpWebRequest 的 BeginGetResponse 方法时，将发出异步网络请求。这个方法返回一个 IAsyncResult，它是 FromAsync 方法的第一个参数。第二个参数是对 EndGetResponse 方法的引用，它需要一个带有 IAsyncResult 参数 (即 EndGetResponse 方法)的委托。第二个参数还需要返回 WebResponse，由 FromAsync 方法的泛型参数决定。当 IAsyncResult 信号完成时，任务助手功能会调用 EndGetResponse 方法(代码文件 Foundations/Program.cs)：

```
private static async void ConvertingAsyncPattern()
{
  HttpWebRequest request = WebRequest.Create("http://www.microsoft.com")
    as HttpWebRequest;

  using (WebResponse response = await Task.Factory.FromAsync<WebResponse>(
    request.BeginGetResponse(null, null), request.EndGetResponse))
  {
    Stream stream = response.GetResponseStream();
    using (var reader = new StreamReader(stream))
    {
      string content = reader.ReadToEnd();
      Console.WriteLine(content.Substring(0, 100));
    }
  }
}
```

**警告：**

在旧应用程序中，通常在使用异步模式时使用委托的 BeginInvoke()方法。在.NET Core 应用程序中使用此方法时，编译器不会报错。但是，在运行时，将抛出一个平台不支持的异常。

## 15.4 错误处理

第 14 章详细介绍了错误和异常处理。但是，在使用异步方法时，应该知道错误的一些特殊处理方式。

所有 ErrorHandling 示例的代码都使用了如下名称空间：

System

System.Threading.Tasks

从一个简单的方法开始，它在延迟后抛出一个异常(代码文件 ErrorHandling/Program.cs)：

```
static async Task ThrowAfter(int ms, string message)
{
  await Task.Delay(ms);
  throw new Exception(message);
}
```

如果调用异步方法，并且没有等待，可以将异步方法放在 try/catch 块中，就会捕获不到异常。这是因为 DontHandle 方法在 ThrowAfter 抛出异常之前，已经执行完毕。需要等待 ThrowAfter 方法，如下一节的示例所示。注意这个代码片段不会抛出异常：

```
private static void DontHandle()
{
  try
  {
    ThrowAfter(200, "first");
    // exception is not caught because this method is finished
    // before the exception is thrown
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

**警告：**

返回 void 的异步方法不会等待。这是因为从 async void 方法抛出的异常无法捕获。因此，异步方法最好返回一个 Task 类型。处理程序方法或重写的基类方法不受此规则限制。

### 15.4.1　异步方法的异常处理

异步方法异常的一个较好处理方式是使用 await 关键字，将其放在 try/catch 语句中，如以下代码块所示。异步调用 ThrowAfter 方法后，HandleOneError 方法就会释放线程，但它会在任务完成时保持任务的引用。此时(2s 后，抛出异常)，会调用匹配的 catch 块内的代码(代码文件 ErrorHandling/Program.cs)。

```
private static async void HandleOneError()
{
  try
  {
    await ThrowAfter(2000, "first");
  }
  catch (Exception ex)
  {
    Console.WriteLine($"handled {ex.Message}");
  }
}
```

### 15.4.2　多个异步方法的异常处理

如果调用两个异步方法，每个都会抛出异常，该如何处理呢？在下面的示例中，第一个 ThrowAfter 方法被调用，2s 后抛出异常(含消息 first)。该方法结束后，另一个 ThrowAfter 方法也被调用，1s 后也抛出异常。事实并非如此，因为对第一个 ThrowAfter 方法的调用已经抛出了异常，try 块内的代码没有继续调用第二个 ThrowAfter 方法，而是在 catch 块内对第一个异常进行处理(代码文件 ErrorHandling/Program.cs)。

```
private static async void StartTwoTasks()
{
  try
  {
    await ThrowAfter(2000, "first");
    await ThrowAfter(1000, "second"); // the second call is not invoked
    // because the first method throws
    // an exception
  }
  catch (Exception ex)
  {
    Console.WriteLine($"handled {ex.Message}");
  }
}
```

现在，并行调用这两个 ThrowAfter 方法。第一个 ThrowAfter 方法 2s 后抛出异常，1s 后第二个 ThrowAfter 方法也抛出异常。使用 Task.WhenAll，不管任务是否抛出异常，都会等到两个任务完成。因此，等待 2s 后，Task.WhenAll 结束，异常被 catch 语句捕获到。但是，只能看见传递给 WhenAll 方法的第一个任务的异常信息。没有显示先抛出异常的任务(第二个任务)，但该任务也在列表中：

```
private async static void StartTwoTasksParallel()
{
  try
  {
    Task t1 = ThrowAfter(2000, "first");
    Task t2 = ThrowAfter(1000, "second");
    await Task.WhenAll(t1, t2);
  }
  catch (Exception ex)
  {
    // just display the exception information of the first task
    // that is awaited within WhenAll
    Console.WriteLine($"handled {ex.Message}");
  }
}
```

有一种方式可以获取所有任务的异常信息，就是在 try 块外声明任务变量 t1 和 t2，使它们可以在 catch 块内访问。在这里，可以使用 IsFaulted 属性检查任务的状态，以确认它们是否为出错状态。若出现异常，IsFaulted 属性会返回 true。可以使用 Task 类的 Exception.InnerException 访问异常信息本身。另一种获取所有任务的异常信息的更好方式如下所述。

### 15.4.3 使用 AggregateException 信息

为了得到所有失败任务的异常信息，可以将 Task.WhenAll 返回的结果写到一个 Task 变量中。这个任务会一直等到所有任务都结束。否则，仍然可能错过抛出的异常。上一小节中，catch 语句只检索到第一个任务的异常。不过，现在可以访问外部任务的 Exception 属性了。Exception 属性是 AggregateException 类型的。这个异常类型定义了 InnerExceptions 属性(不只是 InnerException)，它包含了等待中的所有异常的列表。现在，可以轻松遍历所有异常了(代码文件 ErrorHandling/Program.cs)。

```
private static async void ShowAggregatedException()
{
  Task taskResult = null;
  try
  {
    Task t1 = ThrowAfter(2000, "first");
    Task t2 = ThrowAfter(1000, "second");
    await (taskResult = Task.WhenAll(t1, t2));
  }
  catch (Exception ex)
  {
    Console.WriteLine($"handled {ex.Message}");
    foreach (var ex1 in taskResult.Exception.InnerExceptions)
    {
      Console.WriteLine($"inner exception {ex1.Message}");
    }
  }
}
```

## 15.5 异步与 Windows 应用程序

把 async 关键字用于 UWP 应用程序与本章前面的相同。但需要注意，在 UI 线程中调用 await 之后，当异步方法返回时，将默认返回到 UI 线程中。这便于在异步方法完成后更新 UI 元素。

**注意：**
为了创建 UWP 应用程序，需要 Windows 10，Windows 系统必须在“开发人员模式”下配置。启用“开发人员模式”时，需要打开 Windows 设置，选择 Update & Security 磁贴，选择 For developers 类别，并单击单选

按钮 Developer mode。这样系统就可以运行旁路的应用程序了(未从 Windows Store 中安装的应用程序)，并为“开发人员模式”添加一个 Windows 包。

为了理解功能和问题，创建一个通用 Windows 应用程序。这个应用程序包含 5 个按钮和一个 TextBlock 元素，来演示不同的场景(代码文件 AsyncWindowsApps/MainPage.xaml)：

```
<StackPanel>
  <Button Content="Start Async" Click="OnStartAsync" Margin="4" />
  <Button Content="Start Async with Config" Click="OnStartAsyncConfigureAwait"
    Margin="4" />
  <Button Content="Start Async with Thread Switch"
    Click="OnStartAsyncWithThreadSwitch" Margin="4" />
  <Button Content="Use IAsyncOperation" Click="OnIAsyncOperation" Margin="4" />
  <Button Content="Deadlock" Click="OnStartDeadlock" Margin="4" />
  <TextBlock x:Name="text1" Margin="4" />
</StackPanel>
```

**注意：**
UWP 应用程序在第 33 章到第 36 章详细介绍。

在 OnStartAsync 方法中，UI 线程的线程 ID 写入 TextBlock 元素。接下来调用异步方法 Task.Delay，它不阻塞 UI 线程，在此方法完成后，线程 ID 将再次写入 TextBlock(代码文件 AsyncWindowsApps/MainPage.xaml.cs)：

```
private async void OnStartAsync(object sender, RoutedEventArgs e)
{
  text1.Text = $"UI thread: {GetThread()}";
  await Task.Delay(1000);
  text1.Text += $"\n after await: {GetThread()}";
}
```

为了访问线程 ID，可以使用 Environment 类。在 UWP 应用程序中，Thread 类是无效的——至少在构建版本 15063 之前是这样的：

```
private string GetThread() => $"thread: {Environment.CurrentManagedThreadId}";
```

运行应用程序时，可以在文本元素中看到类似的输出。与控制台应用程序相反，UWP 应用程序定义了一个同步上下文，在等待之后，可以看到与以前相同的线程。这允许直接访问 UI 元素：

```
UI thread: thread 3
after await: thread 3
```

### 15.5.1 配置 await

如果不需要访问 UI 元素，就可以配置 await，以避免使用同步上下文。下面的代码片段演示了配置，并说明为什么不应该从后台线程上访问 UI 元素。

使用 OnStartAsyncConfigureAwait 方法，在将 UI 线程的 ID 写入文本输入后，将调用本地函数 AsyncFunction。在这个本地函数中，启动线程是在调用异步方法 Task.Delay 之前写入的。使用此方法返回的任务，将调用 ConfigureAwait。在这个方法中，任务的配置是通过传递设置为 false 的 continueOnCapturedContext 参数来完成的。通过这种上下文配置，会发现等待之后的线程不再是 UI 线程。使用不同的线程将结果写入 result 变量即可。如 try 块所示，千万不要从非 UI 线程中访问 UI 元素。得到的异常包含 HRESULT 值，如 when 子句所示。只有这个异常在 catch 中捕获：结果返回给调用者。对于调用方，也调用了 ConfigureAwait，但是这次，continueOnCapturedContext 设置为 true。在这里，在等待之前和之后，方法都在 UI 线程中运行(代码文件 AsyncWindowsApp/MainWindow. xaml.cs)：

```
private async void OnStartAsyncConfigureAwait(object sender, RoutedEventArgs e)
{
  text1.Text = $"UI thread: {GetThread()}";

  string s = await AsyncFunction().ConfigureAwait(
    continueOnCapturedContext: true);

  // after await, with continueOnCapturedContext true we are back in the UI thread
  text1.Text += $"\n{s}\nafter await: {GetThread()}";
```

```
    async Task<string> AsyncFunction()
    {
      string result = $"\nasync function: {GetThread()}\n";
      await Task.Delay(1000).ConfigureAwait(continueOnCapturedContext: false);
      result += $"\nasync function after await : {GetThread()}";

      try
      {
        text1.Text = "this is a call from the wrong thread";
        return "not reached";
      }
      catch (Exception ex) when (ex.HResult == -2147417842)
      {
        return result;
        // we know it's the wrong thread
        // don't access UI elements from the previous try block
      }
    }
  }
```

**注意：**
异常处理和过滤在第 14 章中解释。

运行应用程序时，可以看到如下输出。在等待后的异步本地函数中，使用了另一个线程。文本 not reached 从来没有写过，因为抛出了异常：

```
UI thread: thread 3
async function: thread 3
async function after await: thread 6
after await: thread 3
```

**警告：**
在本书后面的 UWP 章节中，使用了数据绑定，而不是直接访问 UI 元素的属性。但是，在 UWP 中，也不能在非 UI 线程中编写绑定到 UI 元素的属性。

## 15.5.2 切换到 UI 线程

在某些情况下，使用后台线程访问 UI 元素并不容易。在这里，可以使用从 Dispatcher 属性返回的 CoreDispatcher 对象切换到 UI 线程。Dispatcher 属性在 DependencyObject 类中定义。DependencyObject 是 UI 元素的基类。调用 CoreDispatcher 对象的 RunAsync 方法会在 UI 线程中再次运行传递进来的 lambda 表达式(代码文件 AsyncWindowsApp/MainWindow.xaml.cs)：

```
private async void OnStartAsyncWithThreadSwitch(object sender, RoutedEventArgs e)
{
  text1.Text = $"UI thread: {GetThread()}";

  string s = await AsyncFunction();

  text1.Text += $"\nafter await: {GetThread()}";

  async Task<string> AsyncFunction()
  {
    string result = $"\nasync function: {GetThread()}\n";
    await Task.Delay(1000).ConfigureAwait(continueOnCapturedContext: false);
    result += $"\nasync function after await : {GetThread()}";

    await Dispatcher.RunAsync(CoreDispatcherPriority.Normal, () =>
    {
      text1.Text +=
        $"\nasync function switch back to the UI thread: {GetThread()}";
    });
    return result;
  }
}
```

运行应用程序时，可以看到在使用 RunAsync 时总是使用的 UI 线程。

```
UI Thread: thread 3
```

```
async function switch back to the UI thread: thread 3
async function: thread 3
async function after await: thread 5
after await: thread 3
```

### 15.5.3　使用 IAsyncOperation

异步方法由 Windows 运行库定义，不返回 Task 或 ValueTask。Task 和 ValueTask 不是 Windows 运行库的一部分。相反，这些方法返回一个实现接口 IAsyncOperation 的对象， IAsyncOperation 并没有根据需要通过 await 关键字来定义方法 GetAwaiter。但是使用 await 关键字时，IAsyncOperation 会自动转换为 Task。还可以使用 AsTask 扩展方法将 IAsyncOperation 对象转换为任务。

在示例应用程序的方法 OnIAsyncOperation 中，调用 MessageDialog 的 ShowAsync 方法。该方法返回一个 IAsyncOperation，可以简单地使用 await 关键字获取结果(代码文件 AsyncWindowsApp/MainWindow.xaml.cs)：

```
private async void OnIAsyncOperation(object sender, RoutedEventArgs e)
{
  var dlg = new MessageDialog("Select One, Two, Or Three", "Sample");

  dlg.Commands.Add(new UICommand("One", null, 1));
  dlg.Commands.Add(new UICommand("Two", null, 2));
  dlg.Commands.Add(new UICommand("Three", null, 3));

  IUICommand command = await dlg.ShowAsync();

  text1.Text = $"Command {command.Id} with the label {command.Label} invoked";
}
```

### 15.5.4　避免阻塞情况

在 Task 上一起使用 Wait 和 async 关键字是很危险的。在使用同步化上下文的应用程序中，这很容易导致死锁。

在方法 OnStartDeadlock 中，调用本地函数 DelayAsync。DelayAsync 等待 Task.Delay 的完成，之后在前台线程中继续执行。但是，调用者在 DelayAsync 返回的任务上调用 Wait()方法。Wait()方法阻塞调用线程，直到任务完成。在这种情况下，Wait()是从前台线程上调用的，因此 Wait()会阻塞前台线程。Task.Delay 上的 Wait()永远无法完成，因为前台线程不可用。这是一个经典的死锁场景(代码文件 AsyncWindowsApps/MainWindow.xaml.cs)：

```
private void OnStartDeadlock(object sender, RoutedEventArgs e)
{
  DelayAsync().Wait();
}

private async Task DelayAsync()
{
  await Task.Delay(1000);
}
```

**警告：**
避免在使用同步化上下文的应用程序中同时使用 Wait 和 await。

## 15.6　小结

本章介绍了 async 和 await 关键字。通过几个示例，介绍了基于任务的异步模式，比.NET 早期版本中的异步模式和基于事件的异步模式更具优势。

本章也讨论了在 Task 类的辅助下，创建异步方法是非常容易的。同时，学会了如何使用 async 和 await 关键字等待这些方法，而不会阻塞线程。最后，介绍了异步方法的错误处理。

若想了解更多关于并行编程、线程和任务的详细信息，请参考第 21 章。

第 16 章将继续关注 C#和.NET 的核心功能，详细介绍了反射、元数据和动态编程。

# 第16章

# 反射、元数据和动态编程

**本章要点**

- 使用自定义特性
- 在运行期间使用反射检查元数据
- 从支持反射的类中构建访问点
- 使用动态类型
- 用 DynamicObject 和 ExpandoObject 创建动态对象

**本章源代码下载地址(wrox.com)：**

打开 www.wrox.com 的 Download Code 选项卡可下载本章源代码。源代码也可以在 ReflectionAndDynamic 目录的 https://github.com/ProfessionalCSharp/ProfessionalCSharp7 中找到。本章代码分为以下几个主要的示例文件：

- LookupWhatsNew
- TypeView
- VectorClass
- WhatsNewAttributes
- Dynamic
- DynamicFileReader

## 16.1 在运行期间检查代码和动态编程

本章讨论自定义特性、反射和动态编程。自定义特性允许把自定义元数据与程序元素关联起来。这些元数据是在编译过程中创建的，并嵌入到程序集中。反射是一个普通术语，它描述了在运行过程中检查和处理程序元素的功能。例如，反射允许完成以下任务：

- 枚举类型的成员
- 实例化新对象
- 执行对象的成员
- 查找类型的信息
- 查找程序集的信息

- 检查应用于某种类型的自定义特性
- 创建和编译新程序集

这个列表列出了许多功能，包括.NET Framework 类库提供的一些最强大、最复杂的功能。但本章不可能介绍反射的所有功能，仅讨论最常用的功能。

为了说明自定义特性和反射，我们将开发一个示例，说明公司如何定期升级软件，自动记录升级的信息。在这个示例中，要定义几个自定义特性，表示程序元素最后修改的日期，以及发生了什么变化。然后使用反射开发一个应用程序，它在程序集中查找这些特性，自动显示软件自某个给定日期以来升级的所有信息。

本章要讨论的另一个示例是一个应用程序，该程序从数据库中读取信息或把信息写入数据库，并使用自定义特性，把类和属性标记为对应的数据库表和列。然后在运行期间从程序集中读取这些特性，使程序可以自动从数据库的相应位置检索或写入数据，不需要为每个表或每一列编写特定的逻辑。

本章的第二部分是动态编程，C#自从第 4 版添加了 dynamic 类型后，动态编程就成为 C#的一部分。随着 Ruby、Python 等语言的成长，以及 JavaScript 的使用更加广泛，动态编程引起了人们越来越多的兴趣。尽管 C#仍是一种静态的类型化语言，但这些新增内容给它提供了一些开发人员期望的动态功能。使用动态语言功能，允许在 C#中调用脚本函数，简化 COM 交互操作。

本章介绍 dynamic 类型及其使用规则，并讨论 DynamicObject 的实现方式和使用方式。另外，还将介绍 DynamicObject 的框架实现方式，即 ExpandoObject。

## 16.2　自定义特性

前面介绍了如何在程序的各个数据项上定义特性。这些特性都是 Microsoft 定义好的，作为.NET Framework 类库的一部分，许多特性都得到了 C#编译器的支持。对于这些特殊的特性，编译器可以以特殊的方式定制编译过程，例如，可以根据 StructLayout 特性中的信息在内存中布置结构。

.NET Framework 也允许用户定义自己的特性。显然，这些特性不会影响编译过程，因为编译器不能识别它们，但这些特性在应用于程序元素时，可以在编译好的程序集中用作元数据。

这些元数据在文档说明中非常有用。但是，使自定义特性非常强大的因素是使用反射，代码可以读取这些元数据，使用它们在运行期间做出决策。也就是说，自定义特性可以直接影响代码运行的方式。例如，自定义特性可以用于支持对自定义许可类进行声明性的代码访问安全检查，把信息与程序元素关联起来，程序元素由测试工具使用，或者在开发可扩展的架构时，允许加载插件或模块。

### 16.2.1　编写自定义特性

为了理解编写自定义特性的方式，应了解一下在编译器遇到代码中某个应用了自定义特性的元素时，该如何处理。以数据库为例，假定有一个 C#属性声明，如下所示。

```
[FieldName("SocialSecurityNumber")]
public string SocialSecurityNumber
{
  get {
    //...
```

当 C#编译器发现这个属性(property)应用了一个 FieldName 特性时，首先会把字符串 Attribute 追加到这个名称的后面，形成一个组合名称 FieldNameAttribute，然后在其搜索路径的所有名称空间(即在 using 语句中提及的名称空间)中搜索有指定名称的类。但要注意，如果用一个特性标记数据项，而该特性的名称以字符串 Attribute 结尾，编译器就不会把该字符串加到组合名称中，而是不修改该特性名。因此，上面的代码等价于：

```
[FieldNameAttribute("SocialSecurityNumber")]
public string SocialSecurityNumber
{
  get {
    //...
```

编译器会找到含有该名称的类，且这个类直接或间接派生自 System.Attribute。编译器还认为这个类包含控

制特性用法的信息。特别是属性类需要指定：

- 特性可以应用到哪些类型的程序元素上(类、结构、属性和方法等)
- 它是否可以多次应用到同一个程序元素上
- 特性在应用到类或接口上时，是否由派生类和接口继承
- 这个特性有哪些必选和可选参数

如果编译器找不到对应的特性类，或者找到一个特性类，但使用特性的方式与特性类中的信息不匹配，编译器就会产生一个编译错误。例如，如果特性类指定该特性只能应用于类，但我们把它应用到结构定义上，就会产生一个编译错误。

继续上面的示例，假定定义了一个 FieldName 特性：

```
[AttributeUsage(AttributeTargets.Property,
  AllowMultiple=false, Inherited=false)]
public class FieldNameAttribute: Attribute
{
  private string _name;
  public FieldNameAttribute(string name)
  {
    _name = name;
  }
}
```

下面几节讨论这个定义中的每个元素。

### 1. 指定 AttributeUsage 特性

要注意的第一个问题是特性(attribute)类本身用一个特性——System.AttributeUsage 特性来标记。这是 Microsoft 定义的一个特性，C#编译器为它提供了特殊的支持(你可能认为 AttributeUsage 根本不是一个特性，它更像一个元特性，因为它只能应用到其他特性上，不能应用到类上)。AttributeUsage 主要用于标识自定义特性可以应用到哪些类型的程序元素上。这些信息由它的第一个参数给出，该参数是必选的，其类型是枚举类型 AttributeTargets。在上面的示例中，指定 FieldName 特性只能应用到属性(property)上——这是因为在前面的代码段中把它应用到属性上。AttributeTargets 枚举的成员如下：

- All
- Assembly
- Class
- Constructor
- Delegate
- Enum
- Event
- Field
- GenericParameter
- Interface
- Method
- Module
- Parameter
- Property
- ReturnValue
- Struct

这个列表列出了可以应用该特性的所有程序元素。注意在把特性应用到程序元素上时，应把特性放在元素前面的方括号中。但是，在上面的列表中，有两个值不对应于任何程序元素：Assembly 和 Module。特性可以应用到整个程序集或模块中，而不是应用到代码中的一个元素上，在这种情况下，这个特性可以放在源代码的任何地方，但需要用关键字 Assembly 或 Module 作为前缀：

```
[assembly:SomeAssemblyAttribute(Parameters)]
[module:SomeAssemblyAttribute(Parameters)]
```

在指定自定义特性的有效目标元素时，可以使用按位 OR 运算符把这些值组合起来。例如，如果指定 FieldName 特性可以同时应用到属性和字段上，可以编写下面的代码：

```
[AttributeUsage(AttributeTargets.Property | AttributeTargets.Field,
  AllowMultiple=false, Inherited=false)]
public class FieldNameAttribute: Attribute
```

也可以使用 AttributeTargets.All 指定自定义特性可以应用到所有类型的程序元素上。AttributeUsage 特性还包含另外两个参数：AllowMultiple 和 Inherited。它们用不同的语法来指定：<ParameterName>=<ParameterValue>，而不是只给出这些参数的值。这些参数是可选的，根据需要，可以忽略它们。

AllowMultiple 参数表示一个特性是否可以多次应用到同一项上，这里把它设置为 false，表示如果编译器遇到下述代码，就会产生一个错误：

```
[FieldName("SocialSecurityNumber")]
[FieldName("NationalInsuranceNumber")]
public string SocialSecurityNumber
{
  //...
```

如果把 Inherited 参数设置为 true，就表示应用到类或接口上的特性也可以自动应用到所有派生的类或接口上。如果特性应用到方法或属性上，它就可以自动应用到该方法或属性等的重写版本上。

### 2. 指定特性参数

下面介绍如何指定自定义特性接受的参数。在编译器遇到下述语句时：

```
[FieldName("SocialSecurityNumber")]
public string SocialSecurityNumber
{
  //...
```

编译器会检查传递给特性的参数(在本例中，是一个字符串)，并查找该特性中带这些参数的构造函数。如果编译器找到一个这样的构造函数，编译器就会把指定的元数据传递给程序集。如果编译器找不到，就生成一个编译错误。如后面所述，反射会从程序集中读取元数据(特性)，并实例化它们表示的特性类。因此，编译器需要确保存在这样的构造函数，才能在运行期间实例化指定的特性。

在本例中，仅为 FieldNameAttribute 类提供一个构造函数，而这个构造函数有一个字符串参数。因此，在把 FieldName 特性应用到一个属性上时，必须为它提供一个字符串作为参数，如上面的代码所示。

如果可以选择特性提供的参数类型，就可以提供构造函数的不同重载方法，尽管一般是仅提供一个构造函数，使用属性来定义任何其他可选参数，下面将介绍可选参数。

### 3. 指定特性的可选参数

在 AttributeUsage 特性中，可以使用另一种语法，把可选参数添加到特性中。这种语法指定可选参数的名称和值，它通过特性类中的公共属性或字段起作用。例如，假定修改 SocialSecurityNumber 属性的定义，如下所示：

```
[FieldName("SocialSecurityNumber", Comment="This is the primary key field")]
public string SocialSecurityNumber { get; set; }
{
  //...
```

在本例中，编译器识别第二个参数的语法<ParameterName>=<ParameterValue>，并且不会把这个参数传递给 FieldNameAttribute 类的构造函数，而是查找一个有该名称的公共属性或字段(最好不要使用公共字段，所以一般情况下要使用特性)，编译器可以用这个属性设置第二个参数的值。如果希望上面的代码工作，就必须给 FieldNameAttribute 类添加一些代码：

```
[AttributeUsage(AttributeTargets.Property,
  AllowMultiple=false, Inherited=false)]
public class FieldNameAttribute : Attribute
{
```

```
    public string Comment { get; set; }
    private string _fieldName;
    public FieldNameAttribute(string fieldName)
    {
      _fieldName = fieldname;
    }
    //...
}
```

## 16.2.2 自定义特性示例：WhatsNewAttributes

本节开始编写前面描述过的示例 WhatsNewAttributes，该示例提供了一个特性，表示最后一次修改程序元素的时间。这个示例比前面所有的示例都复杂，因为它包含 3 个不同的程序集：

- WhatsNewAttributes 程序集——它包含特性的定义。
- VectorClass 程序集——它包含所应用的特性的代码。
- LookUpWhatsNew 程序集——它包含显示已改变的数据项详细信息的项目。

其中，只有 LookUpWhatsNew 程序集是目前为止使用的一个控制台应用程序，其余两个程序集都是库，它们都包含类的定义，但都没有程序的入口点。

### 1. WhatsNewAttributes 库程序集

首先从.NET 标准库的核心 WhatsNewAttributes 程序集开始。其源代码包含在 WhatsNewAttributes.cs 文件中，该文件位于本章示例代码中 WhatsNewAttributes 解决方案的 WhatsNewAttributes 项目中。

WhatsNewAttributes.cs 文件定义了两个特性类：LastModifiedAttribute 和 SupportsWhatsNewAttribute。LastModifiedAttribute 特性可以用于标记最后一次修改数据项的时间，它有两个必选参数(这两个参数传递给构造函数)：修改的日期和包含描述修改信息的字符串。它还有一个可选参数 issues (表示存在一个公共属性)，它可以用来描述该数据项的任何重要问题。

在现实生活中，或许想把特性应用到任何对象上。为了使代码比较简单，这里仅允许将它应用于类、方法和构造函数，并允许它多次应用到同一项上(AllowMultiple=true)，因为可以多次修改某一项，每次修改都需要用一个不同的特性实例来标记。

SupportsWhatsNew 是一个较小的类，它表示不带任何参数的特性。这个特性是一个程序集的特性，它用于把程序集标记为通过 LastModifiedAttribute 维护的文档。这样，以后查看这个程序集的程序会知道，它读取的程序集是我们使用自动文档过程生成的那个程序集。这部分示例的完整源代码如下所示(代码文件 WhatsNewAttributes/WhatsNewAttributes.cs)：

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method |
  AttributeTargets.Constructor, AllowMultiple=true, Inherited=false)]
public class LastModifiedAttribute: Attribute
{
  private readonly DateTime _dateModified;
  private readonly string _changes;
  public LastModifiedAttribute(string dateModified, string changes)
  {
    _dateModified = DateTime.Parse(dateModified);
    _changes = changes;
  }

  public DateTime DateModified => _dateModified;

  public string Changes => _changes;

  public string Issues { get; set; }
}

[AttributeUsage(AttributeTargets.Assembly)]
public class SupportsWhatsNewAttribute: Attribute
{
}
```

根据前面的讨论，这段代码应该相当清楚。不过请注意，属性 DateModified 和 Changes 是只读的。使用表

达式语法，编译器会创建 get 访问器。不需要 set 访问器，因为必须在构造函数中把这些参数设置为必选参数。需要 get 访问器，以便可以读取这些特性的值。

### 2. VectorClass 库

.NET 标准库 VectorClass 引用了 WhatsNewAttributes 库，添加 using 声明后，全局程序集特性标记程序集，以支持 WhatsNew 特性(代码文件 VectorClass/Vector.cs)：

```
[assembly: SupportsWhatsNew]
```

VectorClass 示例代码使用如下名称空间：

System
System.Collections
System.Collections.Generic
WhatsNewAttributes

下面是 Vector 类的代码。给类添加了一些 LastModified 特性，以标记更改：

```
[LastModified("19 Jul 2017", "updated for C# 7 and .NET Core 2")]
[LastModified("6 Jun 2015", "updated for C# 6 and .NET Core")]
[LastModified("14 Deb 2010", "IEnumerable interface implemented: " +
 "Vector can be treated as a collection")]
[LastModified("10 Feb 2010", "IFormattable interface implemented " +
 "Vector accepts N and VE format specifiers")]
public class Vector : IFormattable, IEnumerable<double>
{
  public Vector(double x, double y, double z)
  {
    X = x;
    Y = y;
    Z = z;
  }

  [LastModified("19 Jul 2017", "Reduced the number of code lines")]
  public Vector(Vector vector)
    : this (vector.X, vector.Y, vector.Z { }

  public double X { get; }
  public double Y { get; }
  public double Z { get; }

  public string ToString(string format, IFormatProvider formatProvider)
  {
    //...
```

还标记了被包含的 VectorEnumerator 类：

```
[LastModified("6 Jun 2015",
  "Changed to implement the generic interface IEnumerator<T>")]
[LastModified("14 Feb 2010",
  "Class created as part of collection support for Vector")]
private class VectorEnumerator : IEnumerator<double>
{
```

该库的版本号在 csproj 项目文件中定义(项目文件 VectorClass/VectorClass.csproj)：

```
<PropertyGroup>
  <TargetFramework>netstandard2.0</TargetFramework>
  <Version>2.1.0</Version>
</PropertyGroup>
```

上面是这个示例的代码。目前还不能运行它，因为我们只有两个库。在描述了反射的工作原理后，就介绍这个示例的最后一部分，从中可以查看和显示这些特性。

## 16.3　反射

本节先介绍 System.Type 类，通过这个类可以访问关于任何数据类型的信息。然后简要介绍 System.Reflection.Assembly 类，它可以用于访问给定程序集的相关信息，或者把这个程序集加载到程序中。最后把本节的代码和

上一节的代码结合起来，完成 WhatsNewAttributes 示例。

### 16.3.1 System.Type 类

这里使用 Type 类只为了存储类型的引用：

```
Type t = typeof(double);
```

我们以前把 Type 看作一个类，但它实际上是一个抽象的基类。只要实例化了一个 Type 对象，实际上就实例化了 Type 的一个派生类。尽管一般情况下派生类只提供各种 Type 方法和属性的不同重载，但是这些方法和属性返回对应数据类型的正确数据，Type 有与每种数据类型对应的派生类。它们一般不添加新的方法或属性。通常，获取指向任何给定类型的 Type 引用有 3 种常用方式：

- 使用 C#的 typeof 运算符，如上述代码所示。这个运算符的参数是类型的名称(但不放在引号中)。
- 使用 GetType()方法，所有的类都会从 System.Object 继承这个方法。

```
double d = 10;
Type t = d.GetType();
```

在一个变量上调用 GetType()方法，而不是把类型的名称作为其参数。但要注意，返回的 Type 对象仍只与该数据类型相关：它不包含与该类型的实例相关的任何信息。如果引用了一个对象，但不能确保该对象实际上是哪个类的实例，GetType 方法就很有用。

- 还可以调用 Type 类的静态方法 GetType()：

```
Type t = Type.GetType("System.Double");
```

Type 是许多反射功能的入口。它实现许多方法和属性，这里不可能列出所有的方法和属性，而主要介绍如何使用这个类。注意，可用的属性都是只读的：可以使用 Type 确定数据的类型，但不能使用它修改该类型！

#### 1. Type 的属性

由 Type 实现的属性可以分为下述三类。首先，许多属性都可以获取包含与类相关的各种名称的字符串，如表 16-1 所示。

表 16-1

| 属　　性 | 返　回　值 |
|---|---|
| Name | 数据类型名 |
| FullName | 数据类型的完全限定名(包括名称空间名) |
| Namespace | 在其中定义数据类型的名称空间名 |

其次，属性还可以进一步获取 Type 对象的引用，这些引用表示相关的类，如表 16-2 所示。

表 16-2

| 属　　性 | 返回对应的 Type 引用 |
|---|---|
| BaseType | 该 Type 的直接基本类型 |
| UnderlyingSystemType | 该 Type 在.NET 运行库中映射到的类型(某些.NET 基类实际上映射到由 IL 识别的特定预定义类型)。这个成员只能在完整的框架中使用 |

许多布尔属性表示这种类型是一个类，还是一个枚举等。这些特性包括 IsAbstract、IsArray、IsClass、IsEnum、IsInterface、IsPointer、IsPrimitive(一种预定义的基元数据类型)、IsPublic、IsSealed 以及 IsValueType。例如，使用一种基元数据类型：

```
Type intType = typeof(int);
Console.WriteLine(intType.IsAbstract); // writes false
Console.WriteLine(intType.IsClass); // writes false
Console.WriteLine(intType.IsEnum); // writes false
```

```
Console.WriteLine(intType.IsPrimitive); // writes true
Console.WriteLine(intType.IsValueType); // writes true
```

或者使用 Vector 类：

```
Type vecType = typeof(Vector);
Console.WriteLine(vecType.IsAbstract); // writes false
Console.WriteLine(vecType.IsClass); // writes true
Console.WriteLine(vecType.IsEnum); // writes false
Console.WriteLine(vecType.IsPrimitive); // writes false
Console.WriteLine(vecType.IsValueType); // writes false
```

也可以获取在其中定义该类型的程序集的引用，该引用作为 System.Reflection.Assembly 类的实例的一个引用来返回：

```
Type t = typeof (Vector);
Assembly containingAssembly = new Assembly(t);
```

### 2. 方法

System.Type 的大多数方法都用于获取对应数据类型的成员信息：构造函数、属性、方法和事件等。它有许多方法，但它们都有相同的模式。例如，有两个方法可以获取数据类型的方法的细节信息：GetMethod()和 GetMethods()。GetMethod()方法返回 System.Reflection.MethodInfo 对象的一个引用，其中包含一个方法的细节信息。GetMethods()方法返回这种引用的一个数组。其区别是 GetMethods()方法返回所有方法的细节信息；而 GetMethod()方法返回一个方法的细节信息，其中该方法包含特定的参数列表。这两个方法都有重载方法，重载方法有一个附加的参数，即 BindingFlags 枚举值，该值表示应返回哪些成员，例如，返回公有成员、实例成员和静态成员等。

例如，GetMethods()方法的最简单的一个重载方法不带参数，返回数据类型的所有公共方法的信息：

```
Type t = typeof(double);
foreach (MethodInfo nextMethod in t.GetMethods())
{
  //...
}
```

Type 的成员方法如表 16-3 所示，遵循同一个模式。注意名称为复数形式的方法返回一个数组。

表 16-3

| 返回的对象类型 | 方　法 |
|---|---|
| ConstructorInfo | GetConstructor(),GetConstructors() |
| EventInfo | GetEvent(),GetEvents() |
| FieldInfo | GetField(),GetFields() |
| MemberInfo | GetMember(),GetMembers(),GetDefaultMembers() |
| MethodInfo | GetMethod(),GetMethods() |
| PropertyInfo | GetProperty(),GetProperties() |

GetMember()和 GetMembers()方法返回数据类型的任何成员或所有成员的详细信息，不管这些成员是构造函数、属性和方法等。

## 16.3.2　TypeView 示例

下面用一个短小的示例 TypeView 来说明 Type 类的一些功能，这个示例可以用来列出数据类型的所有成员。本例主要说明对于 double 型 TypeView 的用法，也可以修改该样例中的一行代码，使用其他的数据类型。

运行应用程序的结果输出到控制台上，如下：

```
Analysis of type Double
Type Name: Double
Full Name: System.Double
Namespace: System
```

```
Base Type: ValueType
public members:
System.Double Method IsInfinity
System.Double Method IsPositiveInfinity
System.Double Method IsNegativeInfinity
System.Double Method IsNaN
System.Double Method CompareTo
System.Double Method CompareTo
System.Double Method Equals
System.Double Method op_Equality
System.Double Method op_Inequality
System.Double Method op_LessThan
System.Double Method op_GreaterThan
System.Double Method op_LessThanOrEqual
System.Double Method op_GreaterThanOrEqual
System.Double Method Equals
System.Double Method GetHashCode
System.Double Method ToString
System.Double Method ToString
System.Double Method ToString
System.Double Method ToString
System.Double Method Parse
System.Double Method Parse
System.Double Method Parse
System.Double Method Parse
System.Double Method TryParse
System.Double Method TryParse
System.Double Method GetTypeCode
System.Object Method GetType
System.Double Field MinValue
System.Double Field MaxValue
System.Double Field Epsilon
System.Double Field NegativeInfinity
System.Double Field PositiveInfinity
System.Double Field NaN
```

控制台显示了数据类型的名称、全名和名称空间，以及底层类型的名称。然后，它迭代该数据类型的所有公有实例成员，显示所声明类型的每个成员、成员的类型(方法、字段等)以及成员的名称。声明类型是实际声明类型成员的类的名称(例如，如果在 System.Double 中定义或重载它，该声明类型就是 System.Double，如果成员继承自某个基类，该声明类型就是相关基类的名称)。

TypeView 不会显示方法的签名，因为我们是通过 MemberInfo 对象获取所有公有实例成员的详细信息，参数的相关信息不能通过 MemberInfo 对象来获得。为了获取该信息，需要引用 MemberInfo 和其他更特殊的对象，即需要分别获取每一种类型的成员的详细信息。

TypeView 的示例代码使用如下名称空间：

System

System.Reflection

System.Text

TypeView 会显示所有公有实例成员的详细信息，但对于 double 类型，仅定义了字段和方法。全部代码都放在 Program 一个类中，这个类包含两个静态方法和一个静态字段，StringBuilder 的一个实例称为 OutputText，OutputText 用于创建在消息框中显示的文本。Main()方法和类的声明如下所示(代码文件 TypeView/Program.cs)：

```
class Program
{
  private static StringBuilder OutputText = new StringBuilder();

  static void Main()
  {
    // modify this line to retrieve details of any other data type
    Type t = typeof(double);
    AnalyzeType(t);
    Console.WriteLine($"Analysis of type {t.Name}");
    Console.WriteLine(OutputText.ToString());
    Console.ReadLine();
  }
  //...
}
```

实现的 Main()方法首先声明一个 Type 对象，来表示我们选择的数据类型，再调用方法 AnalyzeType()，

AnalyzeType()方法从 Type 对象中提取信息，并使用该信息构建输出文本。最后在控制台中显示输出。这些都由 AnalyzeType()方法完成：

```
static void AnalyzeType(Type t)
{
  TypeInfo typeInfo = t.GetTypeInfo();
  AddToOutput($"Type Name: {t.Name}");
  AddToOutput($"Full Name: {t.FullName}");
  AddToOutput($"Namespace: {t.Namespace}");

  Type tBase = t.BaseType;
  if (tBase != null)
  {
    AddToOutput($"Base Type: {tBase.Name}");
  }

  AddToOutput("\npublic members:");
  foreach (MemberInfo NextMember in t.GetMembers())
  {
    AddToOutput($"{member.DeclaringType} {member.MemberType} {member.Name}");
  }
}
```

实现 AnalyzeType()方法，仅需要调用 Type 对象的各种属性，就可以获得我们需要的类型名称的相关信息，再调用 GetMembers()方法，获得一个 MemberInfo 对象的数组，该数组用于显示每个成员的信息。注意，这里使用了一个辅助方法 AddToOutput()，该方法创建要显示的文本：

```
static void AddToOutput(string Text) =>
  OutputText.Append("\n" + Text);
```

### 16.3.3　Assembly 类

Assembly 类在 System.Reflection 名称空间中定义，它允许访问给定程序集的元数据，它也包含可以加载和执行程序集(假定该程序集是可执行的)的方法。与 Type 类一样，Assembly 类包含非常多的方法和属性，这里不可能逐一论述。下面仅介绍完成 WhatsNewAttributes 示例所需要的方法和属性。

在使用 Assembly 实例做一些工作前，需要把相应的程序集加载到正在运行的进程中。为此，可以使用静态成员 Assembly.Load()或 Assembly.LoadFrom()。这两个方法的区别是 Load()方法的参数是程序集的名称，运行库会在各个位置上搜索该程序集，试图找到该程序集，这些位置包括本地目录和全局程序集缓存。而 LoadFrom()方法的参数是程序集的完整路径名，它不会在其他位置搜索该程序集：

```
Assembly assembly1 = Assembly.Load("SomeAssembly");
Assembly assembly2 = Assembly.LoadFrom
  (@"C:\My Projects\Software\SomeOtherAssembly");
```

这两个方法都有许多其他重载版本，它们提供了其他安全信息。加载了一个程序集后，就可以使用它的各种属性进行查询，例如，查找它的全名：

```
string name = assembly1.FullName;
```

#### 1. 获取在程序集中定义的类型的详细信息

Assembly 类的一个功能是它可以获得在相应程序集中定义的所有类型的详细信息，只要调用 Assembly.GetTypes()方法，它就可以返回一个包含所有类型的详细信息的 System.Type 引用数组，然后就可以按照上一节的方式处理这些 Type 引用：

```
Type[] types = theAssembly.GetTypes();
foreach(Type definedType in types)
{
  DoSomethingWith(definedType);
}
```

#### 2. 获取自定义特性的详细信息

用于查找在程序集或类型中定义了什么自定义特性的方法取决于与该特性相关的对象类型。如果要确定程序

集从整体上关联了什么自定义特性，就需要调用 Attribute 类的一个静态方法 GetCustomAttributes()，给它传递程序集的引用：

```
Attribute[] definedAttributes =
  Attribute.GetCustomAttributes(assembly1);
// assembly1 is an Assembly object
```

**注意：**

*这是相当重要的。以前你可能想知道，在定义自定义特性时，为什么必须费尽周折为它们编写类，以及为什么 Microsoft 没有更简单的语法。答案就在于此。自定义特性确实与对象一样，加载了程序集后，就可以读取这些特性对象，查看它们的属性，调用它们的方法。*

GetCustomAttributes()方法用于获取程序集的特性，它有两个重载方法：如果在调用它时，除了程序集的引用外，没有指定其他参数，该方法就会返回为这个程序集定义的所有自定义特性。当然，也可以通过指定第二个参数来调用它，第二个参数是表示感兴趣的特性类的一个 Type 对象，在这种情况下，GetCustomAttributes()方法就返回一个数组，该数组包含指定类型的所有特性。

注意，所有特性都作为一般的 Attribute 引用来获取。如果要调用为自定义特性定义的任何方法或属性，就需要把这些引用显式转换为相关的自定义特性类。调用 Assembly.GetCustomAttributes()的另一个重载方法，可以获得与给定数据类型相关的自定义特性的详细信息，这次传递的是一个 Type 引用，它描述了要获取的任何相关特性的类型。另一方面，如果要获得与方法、构造函数和字段等相关的特性，就需要调用 GetCustomAttributes()方法，该方法是 MethodInfo、ConstructorInfo 和 FieldInfo 等类的一个成员。

如果只需要给定类型的一个特性，就可以调用 GetCustomAttribute()方法，它返回一个 Attribute 对象。在 WhatsNewAttributes 示例中使用 GetCustomAttribute()方法，是为了确定程序集中是否有 SupportsWhatsNew 特性。为此，调用 GetCustomAttribute()方法，传递对 WhatsNewAttributes 程序集的一个引用和 SupportsWhatsNewAttribute 特性的类型。如果有这个特性，就返回一个 Attribute 实例。如果在程序集中没有定义任何实例，就返回 null。如果找到两个或多个实例，GetCustomAttribute()方法就抛出一个 System.Reflection.AmbiguousMatchException 异常。该调用如下所示：

```
Attribute supportsAttribute =
  Attribute.GetCustomAttributes(assembly1, typeof(SupportsWhatsNewAttribute));
```

### 16.3.4 完成 WhatsNewAttributes 示例

现在已经有足够的知识来完成 WhatsNewAttributes 示例了。为该示例中的最后一个程序集 LookupWhatsNew 编写源代码，这部分应用程序是一个控制台应用程序，它需要引用其他两个程序集 WhatsNewAttributes 和 VectorClass。

LookupWhatsNew 项目的示例代码引用了 WhatsNewAttributes 和 VectorClass 库，使用了如下名称空间：

System
System.Collections.Generic
System.Linq
System.Reflection
System.Text
WhatsNewAttributes

Program 类包含主程序入口点和其他方法。我们定义的所有方法都在这个类中，它还有两个静态字段：outputText 和 backDateTo。outputText 字段包含在准备阶段创建的文本，这个文本要写到消息框中，backDateTo 字段存储了选择的日期——自从该日期以来进行的所有修改都要显示出来。一般情况下，需要显示一个对话框，让用户选择这个日期，但我们不想编写这种代码，以免转移读者的注意力。因此，把 backDateTo 字段硬编码为日期 2017 年 2 月 1 日。在下载这段代码时，很容易修改这个日期(代码文件 LookupWhatsNew/Program.cs):：

```
class Program
{
  private static readonly StringBuilder outputText = new StringBuilder(1000);
  private static DateTime backDateTo = new DateTime(2017, 2, 1);

  static void Main()
  {
    Assembly theAssembly = Assembly.Load(new AssemblyName("VectorClass"));
    Attribute supportsAttribute = theAssembly.GetCustomAttribute(
      typeof(SupportsWhatsNewAttribute));

    AddToOutput($"Assembly: {theAssembly.FullName}");
    if (supportsAttribute == null)
    {
      AddToOutput("This assembly does not support WhatsNew attributes");
      return;
    }
    else
    {
      AddToOutput("Defined Types:");
    }

    IEnumerable<Type> types = theAssembly.ExportedTypes;
    foreach(Type definedType in types)
    {
      DisplayTypeInfo(definedType);
    }

    Console.WriteLine($"What\`s New since {backDateTo:D}");
    Console.WriteLine(outputText.ToString());
    Console.ReadLine();
  }
  //...
}
```

Main()方法首先加载 VectorClass 程序集，验证它是否用 SupportsWhatsNew 特性标记。我们知道，VectorClass 程序集应用了 SupportsWhatsNew 特性，虽然才编译了该程序集，但进行这种检查还是必要的，因为用户可能希望检查这个程序集。

验证了这个程序集后，使用 Assembly.ExportedTypes 属性获得一个集合，其中包括在该程序集中定义的所有类型，然后在这个集合中遍历它们。对每种类型调用一个方法——DisplayTypeInfo()，它给 outputText 字段添加相关的文本，包括 LastModifiedAttribute 类的任何实例的详细信息。最后，显示带有完整文本的控制台。DisplayTypeInfo()方法如下所示(代码文件 LookupWhatsNew/Program.cs)：

```
private static void DisplayTypeInfo(Type type)
{
  // make sure we only pick out classes
  if (!type.GetTypeInfo().IsClass))
  {
    return;
  }

  AddToOutput($"{Environment.NewLine}class {type.Name}");

  IEnumerable<LastModifiedAttribute> lastModifiedAttributes =
    type.GetTypeInfo().GetCustomAttributes()
    .OfType<LastModifiedAttribute>()
    .Where(a => a.DateModified >= backDateTo).ToArray();

  if (attributes.Count() == 0)
  {
    AddToOutput($"\tNo changes to the class {type.Name}" +
      $"{Environment.NewLine}");
  }
  else
  {
    foreach (LastFieldModifiedAttribute attribute in lastModifiedattributes)
    {
      WriteAttributeInfo(attribute);
    }
  }

  AddToOutput("changes to methods of this class:");
```

```
    foreach (MethodInfo method in
      type.GetTypeInfo().DeclaredMembers.OfType<MethodInfo>())
    {
      IEnumerable<LastModifiedAttribute> attributesToMethods =
        method.GetCustomAttributes().OfType<LastModifiedAttribute>()
          .Where(a => a.DateModified >= backDateTo).ToArray();

      if (attributesToMethods.Count() > 0)
      {
        AddToOutput($"{method.ReturnType} {method.Name}()");
        foreach (Attribute attribute in attributesToMethods)
        {
          WriteAttributeInfo(attribute);
        }
      }
    }
  }
```

注意，在这个方法中，首先应检查所传递的 Type 引用是否表示一个类。因为，为了简化代码，指定 LastModified 特性只能应用于类或成员方法，如果该引用不是类(它可能是一个结构、委托或枚举)，那么进行任何处理都是浪费时间。

接着使用 type.GetTypeInfo().GetCustomAttributes()方法确定这个类是否有相关的 LastModifiedAttribute 实例。如果有，就使用辅助方法 WriteAttributeInfo()把它们的详细信息添加到输出文本中。

最后，使用 TypeInfo 类型的 DeclaredMembers 属性遍历这种数据类型的所有成员方法，然后对每个方法进行相同的处理(类似于对类执行的操作)：检查每个方法是否有相关的 LastModifiedAttribute 实例，如果有，就用 WriteAttributeInfo()方法显示它们。

下面的代码显示了 WriteAttributeInfo()方法，它负责确定为给定的 LastModifiedAttribute 实例显示什么文本，注意因为这个方法的参数是一个 Attribute 引用，所以需要先把该引用强制转换为 LastModifiedAttribute 引用。之后，就可以使用最初为这个特性定义的属性获取其参数。在把该特性添加到要显示的文本中之前，应检查特性的日期是否是最近的(代码文件 LookupWhatsNew/Program.cs)：

```
  private static void WriteAttributeInfo(Attribute attribute)
  {
    if (attribute is LastModifiedAttribute lastModifiedAttribute)
    {
      AddToOutput($"\tmodified: {lastModifiedAttribute.DateModified:D}: " +
        $"{lastModifiedAttribute.Changes}");

      if (lastModifiedAttribute.Issues != null)
      {
        AddToOutput($"\tOutstanding issues: {lastModifiedAttribute.Issues}");
      }
    }
  }
```

最后，是辅助方法 AddToOutput ()：

```
  static void AddToOutput(string text) =>
    outputText.Append($"{Environment.NewLine}{text}");
```

运行这段代码，得到如下结果：

```
What`s New since Wednesday, February 1, 2017
Assembly: VectorClass, Version=2.1.0.0, Culture=neutral, PublicKeyToken=null
Defined Types:

class Vector
      modified: Wednesday, July 19, 2017: updated for C# 7 and .NET Core 2
changes to methods of this class:
System.String ToString()
      modified: Wednesday, July 19, 2017: changed ijk format from StringBuilder to format string
```

注意，在列出 VectorClass 程序集中定义的类型时，实际上选择了两个类：Vector 类和内嵌的 VectorEnumerator 类。还要注意，这段代码把 backDateTo 日期硬编码为 2 月 1 日，实际上选择的是日期为 7 月 19 日的特性(添加集合支持的时间)，而不是前述日期。

# 16.4　为反射使用动态语言扩展

前面一直使用反射来读取元数据。还可以使用反射，从编译时还不清楚的类型中动态创建实例。下一个示例显示了创建 Calculator 类的一个实例，而编译器在编译时不知道这种类型。程序集 CalculatorLib 是动态加载的，没有添加引用。在运行期间，实例化 Calculator 对象，调用方法。知道如何使用 ReflectionAPI 后，使用 C# dynamic 关键字可以完成相同的操作。这个关键字自 C# 4 版本以来，就成为 C#语言的一部分。

## 16.4.1　创建 Calculator 库

要加载的库是一个简单的类库，包含 Calculator 类型与 Add 和 Subtract 方法的实现代码。因为方法是很简单的，所以它们使用表达式语法实现(代码文件 CalculatorLib/Calculator.cs)：

```
public class Calculator
{
  public double Add(double x, double y) => x + y;
  public double Subtract(double x, double y) => x - y;
}
```

编译库后，将 DLL 复制到文件夹 c:/addins。

## 16.4.2　动态实例化类型

为了使用反射动态创建 Calculator 实例，应创建一个 Console App (.NET Core)，命名为 ClientApp。

常量 CalculatorTypeName 定义了 Calculator 类型的名称，包括名称空间。Main()方法需要一个命令行参数指定库的路径，然后调用 UsingReflection 和 UsingReflectionWithDynamic 方法，这两个变体进行反射(代码文件 DynamicSamples/ClientApp/Program.cs)：

```
class Program
{
  private const string CalculatorTypeName = "CalculatorLib.Calculator";

  static void Main(string[] args)
  {
    if (args.Length != 1)
    {
      ShowUsage();
      return;
    }
    UsingReflection(args[0]);
    UsingReflectionWithDynamic(args[0]);
  }

  private static void ShowUsage()
  {
    Console.WriteLine($"Usage: {nameof(ClientApp)} path");
    Console.WriteLine();
    Console.WriteLine("Copy CalculatorLib.dll to an addin directory");
    Console.WriteLine("and pass the absolute path of this directory " +
      "when starting the application to load the library");
  }
```

在使用反射调用方法之前，需要实例化 Calculator 类型。GetCalculator 方法使用 Assembly 类的方法 LoadFile 动态加载程序集，并使用 CreateInstance 方法创建一个 Calculator 类型的实例:

```
private static object GetCalculator()
{
  Assembly assembly = Assembly.LoadFile(CalculatorLibPath);
  return assembly.CreateInstance(CalculatorTypeName);
}
```

ClientApp 的示例代码使用了以下依赖项和.NET 名称空间：

**依赖项**

System.Runtime.Loader

**.NET 名称空间**

Microsoft.CSharp.RuntimeBinder

System

System.Reflection

### 16.4.3 用 Reflection API 调用成员

接下来，使用 Reflection API 调用 Calculator 实例的方法 Add()。首先，Calculator 实例使用辅助方法 GetCalculator()来检索。如果想添加对 CalculatorLib 的引用，可以使用 new Calculator 创建一个实例。但这并不是那么容易。

使用反射调用方法的优点是，类型不需要在编译期间可用。只要把库复制到指定的目录中，就可以在稍后添加它。为了使用反射调用成员，利用 GetType 方法检索实例的 Type 对象——它是基类 Object 的方法。通过扩展方法 GetMethod()(这个方法在 NuGet 包 System.Reflection.TypeExtensions 中定义) 访问 MethodInfo 对象的 Add()方法。MethodInfo 定义了 Invoke()方法，使用任意数量的参数调用该方法。Invoke()方法的第一个参数需要调用成员的类型的实例。第二个参数是 object[]类型，传递调用所需的所有参数。这里传递 x 和 y 变量的值(代码文件 DynamicSamples/ClientApp/Program.cs)：

```
private static void UsingReflection()
{
  double x = 3;
  double y = 4;
  object calc = GetCalculator();

  object result = calc.GetType().GetMethod("Add")
    .Invoke(calc, new object[] { x, y });
  Console.WriteLine($"the result of {x} and {y} is {result}");
}
```

运行该程序，调用计算器，结果写入控制台：

```
The result of 3 and 4 is 7
```

动态调用成员有很多工作要做。下一节看看如何使用 dynamic 关键字。

### 16.4.4 使用动态类型调用成员

使用反射和 dynamic 关键字，从 GetCalculator 方法返回的对象分配给一个 dynamic 类型的变量。该方法本身没有改变，它还返回一个对象。结果返回给一个 dynamic 类型的变量。现在，调用 Add 方法，给它传递两个 double 值(代码文件 DynamicSamples/ClientApp/Program.cs)：

```
private static void ReflectionNew()
{
  double x = 3;
  double y = 4;
  dynamic calc = GetCalculator();
  double result = calc.Add(x, y);
  Console.WriteLine($"the result of {x} and {y} is {result}");
}
```

语法很简单，看起来像是用强类型访问方式调用一个方法。然而，Visual Studio 没有提供智能感知功能，因为可以立即在 Visual Studio 编辑器中看到编码，所以很容易出现拼写错误。

也没有在编译时进行检查。调用 Multiply 方法时，编译器运行得很好。只需要记住，定义了计算器的 Add 和 Subtract 方法。

```
try
{
  result = calc.Multiply(x, y);
}
catch (RuntimeBinderException ex)
{
  Console.WriteLine(ex);
}
```

运行应用程序，调用 Multiply 方法，就会得到一个 RuntimeBinderException 异常：

```
Microsoft.CSharp.RuntimeBinder.RuntimeBinderException: 'CalculatorLib.Calculator'
  does not contain a definition for 'Multiply'
  at CallSite.Target(Closure , CallSite , Object , Double , Double )
  at System.Dynamic.UpdateDelegates.UpdateAndExecute3[T0,T1,T2,TRet](CallSite
  site, T0 arg0, T1 arg1, T2 arg2)
  at ClientApp.Program.UsingReflectionWithDynamic(String addinPath) in...
```

与以强类型方式访问对象相比，使用 dynamic 类型也有更多的开销。因此，这个关键字只用于某些特定的情形，如反射。调用 Type 的 InvokeMember 方法没有进行编译器检查，而是给成员名字传递一个字符串。使用 dynamic 类型的语法很简单，与在这样的场景中使用 Reflection API 相比，有很大的优势。

dynamic 类型还可以用于 COM 集成和脚本环境，详细讨论 dynamic 关键字后，会探讨它。

## 16.5　dynamic 类型

dynamic 类型允许编写忽略编译期间的类型检查的代码。编译器假定，给 dynamic 类型的对象定义的任何操作都是有效的。如果该操作无效，则在代码运行之前不会检测该错误，如下面的示例所示：

```
class Program
{
  static void Main()
  {
    var staticPerson = new Person();
    dynamic dynamicPerson = new Person();
    staticPerson.GetFullName("John", "Smith");
    dynamicPerson.GetFullName("John", "Smith");
  }
}

class Person
{
  public string FirstName { get; set; }
  public string LastName { get; set; }
  public string GetFullName() => $"{FirstName} {LastName}";
}
```

这个示例没有编译，因为它调用了 staticPerson.GetFullName()方法。因为 Person 对象上的方法不接受两个参数，所以编译器会提示出错。如果注释掉该行代码，这个示例就会编译。如果执行它，就会发生一个运行错误。所抛出的异常是 RuntimeBinderException 异常。RuntimeBinder 对象会在运行时判断该调用，确定 Person 类是否支持被调用的方法。这将在本章后面讨论。

与 var 关键字不同，定义为 dynamic 的对象可以在运行期间改变其类型。注意在使用 var 关键字时，对象类型的确定会延迟。类型一旦确定，就不能改变。动态对象的类型可以改变，而且可以改变多次，这不同于把对象的类型强制转换为另一种类型。在强制转换对象的类型时，是用另一种兼容的类型创建一个新对象。例如，不能把 int 强制转换为 Person 对象。在下面的示例中，如果对象是动态对象，就可以把它从 int 变成 Person 类型：

```
dynamic dyn;
dyn = 100;
Console.WriteLine(dyn.GetType());
Console.WriteLine(dyn);
dyn = "This is a string";
Console.WriteLine(dyn.GetType());
Console.WriteLine(dyn);
dyn = new Person() { FirstName = "Bugs", LastName = "Bunny" };
Console.WriteLine(dyn.GetType());
Console.WriteLine($"{dyn.FirstName} {dyn.LastName}");
```

执行这段代码可以看出，dyn 对象的类型实际上从 System.Int32 变成 System.String，再变成 Person。如果 dyn 声明为 int 或 string，这段代码就不会编译。

**注意：**

对于 dynamic 类型有两个限制。动态对象不支持扩展方法，匿名函数(lambda 表达式)也不能用作动态方法

调用的参数，因此 LINQ 不能用于动态对象。大多数 LINQ 调用都是扩展方法，而 lambda 表达式用作这些扩展方法的参数。

## 后台上的动态操作

在后台，这些是如何发生的？C#仍是一种静态的类型化语言，这一点没有改变。看看使用 dynamic 类型生成的 IL(中间语言)。

首先，看看下面的示例 C#代码(代码文件 DynamicSamples/DecompileSample/Program.cs)：

```
class Program
{
  static void Main()
  {
    StaticClass staticObject = new StaticClass();
    DynamicClass dynamicObject = new DynamicClass();
    Console.WriteLine(staticObject.IntValue);
    Console.WriteLine(dynamicObject.DynValue);
    Console.ReadLine();
  }
}

class StaticClass
{
  public int IntValue = 100;
}

class DynamicClass
{
  public dynamic DynValue = 100;
}
```

其中有两个类 StaticClass 和 DynamicClass。StaticClass 类有一个返回 int 的字段。DynamicClass 有一个返回 dynamic 对象的字段。Main()方法仅创建了这些对象，并输出方法返回的值。该示例非常简单。

现在注释掉 Main()方法中对 DynamicClass 类的引用：

```
static void Main()
{
  StaticClass staticObject = new StaticClass();
  //DynamicClass dynamicObject = new DynamicClass();
  Console.WriteLine(staticObject.IntValue);
  //Console.WriteLine(dynamicObject.DynValue);
  Console.ReadLine();
}
```

使用 ildasm 工具，可以看到给 Main()方法生成的 IL：

```
.method private hidebysig static void  Main() cil managed
{
  .entrypoint
  // Code size      22 (0x16)
  .maxstack  8
  IL_0000:  newobj     instance void DecompileSample.StaticClass::.ctor()
  IL_0005:  ldfld      int32 DecompileSample.StaticClass::IntValue
  IL_000a:  call       void [System.Console]System.Console::WriteLine(int32)
  IL_000f:  call       string [System.Console]System.Console::ReadLine()
  IL_0014:  pop
  IL_0015:  ret
} // end of method Program::Main
```

这里不讨论 IL 的细节，只看看这段代码，就可以看出其作用。第 0000 行调用了 StaticClass 构造函数，第 0005 行调用了 StaticClass 类的 IntValue 字段。下一行输出了其值。

现在注释掉对 StaticClass 类的引用，取消 DynamicClass 引用的注释：

```
static void Main()
{
  //StaticClass staticObject = new StaticClass();
  DynamicClass dynamicObject = new DynamicClass();
  //Console.WriteLine(staticObject.IntValue);
  Console.WriteLine(dynamicObject.DynValue);
  Console.ReadLine();
}
```

再次编译应用程序，下面是生成的 IL：

```
.method private hidebysig static void  Main() cil managed
{
  .entrypoint
  // Code size       119 (0x77)
  .maxstack  9
  .locals init (class DecompileSample.DynamicClass V_0)
  IL_0000:  newobj     instance void DecompileSample.DynamicClass::.ctor()
  IL_0005:  stloc.0
  IL_0006:  ldsfld     class
[System.Linq.Expressions]System.Runtime.CompilerServices.CallSite`1<class
[System.Runtime]System.Action`3<class
[System.Linq.Expressions]System.Runtime.CompilerServices.CallSite,class
[System.Runtime]System.Type,object>> DecompileSample.Program/'<>o__0'::'<>p__0'
  IL_000b:  brtrue.s   IL_004c
  IL_000d:  ldc.i4     0x100
  IL_0012:  ldstr      „WriteLine"
  IL_0017:  ldnull
  IL_0018:  ldtoken    DecompileSample.Program
  IL_001d:  call       class [System.Runtime]System.Type
[System.Runtime]System.Type::GetTypeFromHandle(valuetype [System.Runtime]System.RuntimeTypeHandle)
  IL_0022:  ldc.i4.2
  IL_0023:  newarr     [Microsoft.CSharp]Microsoft.CSharp.RuntimeBinder.CSharpArgumentInfo
  IL_0028:  dup
  IL_0029:  ldc.i4.0
  IL_002a:  ldc.i4.s   33
  IL_002c:  ldnull
  IL_002d:  call       class [Microsoft.CSharp]Microsoft.CSharp.RuntimeBinder.CSharpArgumentInfo
[Microsoft.CSharp]Microsoft.CSharp.RuntimeBinder.CSharpArgumentInfo::Create(valuetype
[Microsoft.CSharp]Microsoft.CSharp.RuntimeBinder.CSharpArgumentInfoFlags,
    string)
  IL_0032:  stelem.ref
  IL_0033:  dup
  IL_0034:  ldc.i4.1
  IL_0035:  ldc.i4.0
  IL_0036:  ldnull
  IL_0037:  call       class [Microsoft.CSharp]Microsoft.CSharp.RuntimeBinder.CSharpArgumentInfo
[Microsoft.CSharp]Microsoft.CSharp.RuntimeBinder.CSharpArgumentInfo::Create(valuetype
[Microsoft.CSharp]Microsoft.CSharp.RuntimeBinder.CSharpArgumentInfoFlags,
    string)
  IL_003c:  stelem.ref
  IL_003d:  call       class
[System.Linq.Expressions]System.Runtime.CompilerServices.CallSiteBinder
[Microsoft.CSharp]Microsoft.CSharp.RuntimeBinder.Binder::InvokeMember(valuetype
[Microsoft.CSharp]Microsoft.CSharp.RuntimeBinder.CSharpBinderFlags,
    string,
    class [System.Runtime]System.Collections.Generic.IEnumerable`1<class [System.Runtime]System.Type>,
    class [System.Runtime]System.Type,
    class [System.Runtime]System.Collections.Generic.IEnumerable`1<class
[Microsoft.CSharp]Microsoft.CSharp.RuntimeBinder.CSharpArgumentInfo>)
  IL_0042:  call       class
[System.Linq.Expressions]System.Runtime.CompilerServices.CallSite`1<!0> class
[System.Linq.Expressions]System.Runtime.CompilerServices.CallSite`1<class
[System.Runtime]System.Action`3<class
[System.Linq.Expressions]System.Runtime.CompilerServices.CallSite,class
[System.Runtime]System.Type,object>>::Create(class
[System.Linq.Expressions]System.Runtime.CompilerServices.CallSiteBinder)
  IL_0047:  stsfld     class
[System.Linq.Expressions]System.Runtime.CompilerServices.CallSite`1<class
[System.Runtime]System.Action`3<class
[System.Linq.Expressions]System.Runtime.CompilerServices.CallSite,class
[System.Runtime]System.Type,object>> DecompileSample.Program/'<>o__0'::'<>p__0'
  IL_004c:  ldsfld     class
[System.Linq.Expressions]System.Runtime.CompilerServices.CallSite`1<class
[System.Runtime]System.Action`3<class
[System.Linq.Expressions]System.Runtime.CompilerServices.CallSite,class
[System.Runtime]System.Type,object>> DecompileSample.Program/'<>o__0'::'<>p__0'
  IL_0051:  ldfld      !0 class
[System.Linq.Expressions]System.Runtime.CompilerServices.CallSite`1<class
[System.Runtime]System.Action`3<class
[System.Linq.Expressions]System.Runtime.CompilerServices.CallSite,class
[System.Runtime]System.Type,object>>::Target
  IL_0056:  ldsfld     class
[System.Linq.Expressions]System.Runtime.CompilerServices.CallSite`1<class
[System.Runtime]System.Action`3<class
[System.Linq.Expressions]System.Runtime.CompilerServices.CallSite,class
```

```
[System.Runtime]System.Type,object>> DecompileSample.Program/'<>o__0'::'<>p__0'
      IL_005b:  ldtoken    [System.Console]System.Console
      IL_0060:  call       class [System.Runtime]System.Type
    [System.Runtime]System.Type::GetTypeFromHandle(valuetype
[System.Runtime]System.RuntimeTypeHandle)
      IL_0065:  ldloc.0
      IL_0066:  ldfld      object DecompileSample.DynamicClass::DynValue
      IL_006b:  callvirt   instance void class [System.Runtime]System.Action`3<class
[System.Linq.Expressions]System.Runtime.CompilerServices.CallSite,class
[System.Runtime]System.Type,object>::Invoke(!0,
    !1,
    !2)
      IL_0070:  call       string [System.Console]System.Console::ReadLine()
      IL_0075:  pop
      IL_0076:  ret
    } // end of method Program::Main
```

显然，C#编译器做了许多工作，以支持动态类型。在生成的代码中，会看到对 System.Runtime.CompilerServices.CallSite 类和 System.Runtime.CompilerServices. CallSiteBinder 类的引用。

CallSite 是在运行期间处理查找操作的类型。在运行期间调用动态对象时，必须找到该对象，看看其成员是否存在。CallSite 会缓存这个信息，这样查找操作就不需要重复执行。没有这个过程，循环结构的性能就有问题。

CallSite 完成了成员查找操作后，就调用 CallSiteBinder()方法。它从 CallSite 中提取信息，并生成表达式树，来表示绑定器绑定的操作。

显然这需要做许多工作。优化非常复杂的操作时要格外小心。显然，使用 dynamic 类型是有用的，但它是有代价的。

## 16.6 DynamicObject 和 ExpandoObject 概述

如果要创建自己的动态对象，该怎么办？有两种方法：从 DynamicObject 中派生，或者使用 ExpandoObject。使用 DynamicObject 需要做的工作较多，因为必须重写几个方法。ExpandoObject 是一个可立即使用的密封类。

### 16.6.1 DynamicObject

考虑一个表示人的对象。一般应定义名字、中间名和姓氏等属性。现在假定要在运行期间构建这个对象，且系统事先不知道该对象有什么属性或该对象可能支持什么方法。此时就可以使用基于 DynamicObject 的对象。需要这类功能的场合几乎没有，但到目前为止，C#语言还没有提供该功能(代码文件 DynamicSamples/DynamicSample/WroxDyamicObject.cs)。

首先看看 DynamicObject(代码文件 DynamicSamples/DynamicSample/WroxDyamicObject.cs)：

```
public class WroxDynamicObject : DynamicObject
{
  private Dictionary<string, object> _dynamicData =
    new Dictionary<string, object>();

  public override bool TryGetMember(GetMemberBinder binder, out object result)
  {
    bool success = false;
    result = null;
    if (_dynamicData.ContainsKey(binder.Name))
    {
      result = _dynamicData[binder.Name];
      success = true;
    }
    else
    {
      result = "Property Not Found!";
    }
    return success;
  }

  public override bool TrySetMember(SetMemberBinder binder, object value)
  {
    _dynamicData[binder.Name] = value;
    return true;
  }
```

```
    public override bool TryInvokeMember(InvokeMemberBinder binder,
      object[] args, out object result)
    {
      dynamic method = _dynamicData[binder.Name];
      result = method((DateTime)args[0]);
      return result != null;
    }
  }
```

在这个示例中，重写了 3 个方法 TrySetMember()、TryGetMember()和 TryInvokeMember()。

TrySetMember()方法给对象添加了新方法、属性或字段。本例把成员信息存储在一个 Dictionary 对象中。传送给 TrySetMember()方法的 SetMemberBinder 对象包含 Name 属性，它用于标识 Dictionary 中的元素。

TryGetMember()方法根据 GetMemberBinder 对象的 Name 属性检索存储在 Dictionary 中的对象。

下面的代码使用了刚才新建的动态对象(代码文件 DynamicSamples/DynamicSample/Program.cs)：

```
dynamic wroxDyn = new WroxDynamicObject();
wroxDyn.FirstName = "Bugs";
wroxDyn.LastName = "Bunny";
Console.WriteLine(wroxDyn.GetType());
Console.WriteLine($"{wroxDyn.FirstName} {wroxDyn.LastName}");
```

看起来很简单，但在哪里调用了重写的方法？正是.NET 帮助完成了调用。DynamicObject 处理了绑定，我们只需要引用 FirstName 和 LastName 属性即可，就好像它们一直存在一样。

添加方法很简单。可以使用上例中的 WroxDynamicObject，给它添加 GetTomorrowDate()方法。该方法接受一个 DateTime 对象为参数，返回表示第二天的日期字符串。代码如下：

```
dynamic wroxDyn = new WroxDynamicObject();
Func<DateTime, string> GetTomorrow = today => today.AddDays(1).ToShortDateString();
wroxDyn.GetTomorrowDate = GetTomorrow;
Console.WriteLine($"Tomorrow is {wroxDyn.GetTomorrowDate(DateTime.Now)}");
```

这段代码使用 Func<T, TResult>创建了委托 GetTomorrow。该委托表示的方法调用了 AddDays，给传入的 Date 加上一天，返回得到的日期字符串。接着把委托设置为 wroxDyn 对象上的 GetTomorrowDate()方法。最后一行调用新方法，并传递今天的日期。动态功能再次发挥了作用，对象上有了一个有效的方法。

## 16.6.2　ExpandoObject

ExpandoObject 的工作方式类似于上一节创建的 WroxDynamicObject，区别是不必重写方法，如下面的代码示例所示(代码文件 DynamicSamples/DynamicSample/WroxDynamicObject.cs)：

```
static void DoExpando()
{
  dynamic expObj = new ExpandoObject();
  expObj.FirstName = "Daffy";
  expObj.LastName = "Duck";
  Console.WriteLine($"{expObj.FirstName} {expObj.LastName}");
  Func<DateTime, string> GetTomorrow = today =>
    today.AddDays(1).ToShortDateString();

  expObj.GetTomorrowDate = GetTomorrow;
  Console.WriteLine($"Tomorrow is {expObj.GetTomorrowDate(DateTime.Now)}");
  expObj.Friends = new List<Person>();
  expObj.Friends.Add(new Person() { FirstName = "Bob", LastName = "Jones" });
  expObj.Friends.Add(new Person() { FirstName = "Robert",
    LastName = "Jones" });
  expObj.Friends.Add(new Person() { FirstName = "Bobby", LastName = "Jones" });
  foreach (Person friend in expObj.Friends)
  {
    Console.WriteLine($"{friend.FirstName} {friend.LastName}");
  }
}
```

注意，这段代码与前面的代码几乎完全相同，也添加了 FirstName 和 LastName 属性，以及 GetTomorrow 函数，但它还多做了一件事——把一个 Person 对象集合添加为对象的一个属性。

初看起来，这似乎与使用 dynamic 类型没有区别。但其中有两个微妙的区别非常重要。第一，不能仅创建 dynamic 类型的空对象。必须把 dynamic 类型赋予某个对象，例如，下面的代码是无效的：

```
dynamic dynObj;
dynObj.FirstName = "Joe";
```

与前面的示例一样，此时可以使用 ExpandoObject。

第二，因为 dynamic 类型必须赋予某个对象，所以，如果执行 GetType 调用，它就会报告赋予了 dynamic 类型的对象类型。所以，如果把它赋予 int，GetType 就报告它是一个 int。这不适用于 ExpandoObject 或派生自 DynamicObject 的对象。

如果需要控制动态对象中属性的添加和访问，则使该对象派生自 DynamicObject 是最佳选择。使用 DynamicObject，可以重写几个方法，准确地控制对象与运行库的交互方式。而对于其他情况，就应使用 dynamic 类型或 ExpandoObject。

下面是使用 dynamic 类型和 ExpandoObject 的另一个例子。假设需求是开发一个通用的逗号分隔值(CSV)文件的解析工具。从一个扩展到另一个扩展时，不知道文件中将包含什么数据，只知道值之间是用逗号分隔的，并且第一行包含字段名。

首先，打开文件并读入数据流。这可以用一个简单的辅助方法完成(代码文件 DynamicSamples/DynamicFileReader/DynamicFileHelper.cs)：

```
public class DynamicFileHelper
{
  //...
  private StreamReader OpenFile(string fileName)
  {
    if(File.Exists(fileName))
    {
      return new StreamReader(fileName);
    }
    return null;
  }
  //...
}
```

这段代码打开文件，并创建一个新的 StreamReader 来读取文件内容。

接下来要获取字段名。方法很简单：读取文件的第一行，使用 Split 函数创建字段名的一个字符串数组。

```
string[] headerLine = fileStream.ReadLine().Split(',').Trim().ToArray();
```

接下来的部分很有趣：读入文件的下一行，就像处理字段名那样创建一个字符串数组，然后创建动态对象。具体代码如下所示(代码文件 DynamicSamples/DynamicFileReader/DynamicFileHelper.cs)：

```
public class DynamicFileHelper
{
  //...
  public IEnumerable<dynamic> ParseFile(string fileName)
  {
    var retList = new List<dynamic>();
    while (fileStream.Peek() > 0)
    {
      string[] dataLine = fileStream.ReadLine().Split(',').Trim().ToArray();
      dynamic dynamicEntity = new ExpandoObject();
      for(int i=0;i<headerLine.Length;i++)
      {
        ((IDictionary<string,object>)dynamicEntity).Add(headerLine[i],
          dataLine[i]);
      }
      retList.Add(dynamicEntity);
    }
    return retList;
  }
  //...
}
```

有了字段名和数据元素的字符串数组后，创建一个新的 ExpandoObject，在其中添加数据。注意，代码中将 ExpandoObject 强制转换为 Dictionary 对象。用字段名作为键，数据作为值。然后，把这个新对象添加到所创建的 retList 对象中，返回给调用该方法的代码。

这样做的好处是有了一段可以处理传递给它的任何数据的代码。这里唯一的要求是确保第一行是字段名，并且所有的值是用逗号分隔的。可以把这个概念扩展到其他文件类型，甚至 DataReader。

使用这个 CSV 文件内容和下载的示例代码：

```
FirstName, LastName, City, State
Niki, Lauda, Vienna, Austria
Carlos, Reutemann, Santa Fe, Argentine
Sebastian, Vettel, Thurgovia, Switzerland
```

以及 Main()方法，读取示例文件 EmployeeList.txt(代码文件 DynamicSamples/DynamicFileReader/Program.cs)：

```
static void Main()
{
  var helper = new DynamicFileHelper();
  var employeeList = helper.ParseFile("EmployeeList.txt");
  foreach (var employee in employeeList)
  {
    Console.WriteLine($"{employee.FirstName} {employee.LastName} lives in " +
      $"{employee.City}, {employee.State}.");
  }
  Console.ReadLine();
}
```

把如下结果输出到控制台：

```
Niki Lauda lives in Vienna, Austria.
Carlos Reutemann lives in Santa Fe, Argentine.
Sebastian Vettel lives in Thurgovia, Switzerland.
```

## 16.7　小结

本章介绍了 Type 和 Assembly 类，它们是访问反射所提供的扩展功能的主要入口点。

另外，本章还探讨了反射的一个常用方面：自定义特性，它比其他方面更常用。介绍了如何定义和应用自己的自定义特性，以及如何在运行期间检索自定义特性的信息。

本章的第二部分介绍了 dynamic 类型。通过使用 ExpandoObject 代替多个对象，代码量会显著减少。另外，通过使用 DLR 及添加 Python 或 Ruby 等脚本语言，可以创建多态性更好的应用程序，改变它们十分简单，并且不需要重新编译。

下一章将详细介绍如何使用 IDisposable 接口释放原生资源，并使用不安全的 C#代码。

# 第17章 托管和非托管内存

**本章要点**

- 运行期间在栈和堆上分配空间
- 垃圾收集
- 使用析构函数和 System.IDisposable 接口释放非托管的资源
- C#中使用指针的语法
- 引用语义
- 使用 Span 类型
- 平台调用，访问本机 API

**本章源代码下载地址(wrox.com)：**

打开 www.wrox.com 的 Download Code 选项卡可下载本章源代码。源代码也可以在 Memory 目录的 https://github.com/ProfessionalCSharp/ProfessionalCSharp7 中找到。本章代码分为以下几个主要的示例文件：

- PointerPlayground
- PointerPlayground2
- QuickArray
- ReferenceSemantics
- SpanSample
- PlatformInvokeSample

## 17.1 内存

变量存储在堆栈中。它引用的数据可以位于栈(结构)或堆(类)上。结构体也可以装箱，这样对象就会在堆上创建。垃圾收集器需要从托管堆中释放不再需要的非托管对象。使用本机 API，可以在本机堆上分配内存。垃圾收集器不负责在本机堆上分配内存。必须自己释放这些内存。关于内存，有很多东西需要考虑。

使用托管环境时，很容易被误导，注意不到内存管理，因为垃圾收集器(GC)会处理它。很多工作都由 GC 完成；了解它是如何工作的，什么是大小对象堆，以及什么数据类型存储在堆栈上是非常有益的。同时，垃圾收集器处理托管的资源，那么非托管资源呢？它们需要由开发人员释放。程序可能是完全托管的程序，但是框

架的类型呢？例如，文件类型（参见第 22 章）包装了一个本地文件句柄。这个文件句柄需要释放。为了尽早释放这个句柄，最好了解 IDisposable 接口和 using 语句，参见本章的内容。

其他方面也很重要。尽管一些语言结构更易于创建不可变的类型，但可变对象也有优势。string 类是自.NET Framework 1.0 以来一直可用的不可变类型。现在我们经常需要处理大的字符串。在操作字符串时，GC 需要清理许多对象。直接访问字符串的内存并进行更改，将使程序可变，在不同的场景中具有更好的性能。Span 类型使之成为可能，参见第 9 章和第 7 章。对于数组，还介绍了 ArrayPool 类，该类还可以减少 GC 的工作量。

本章介绍内存管理和内存访问的各个方面。如果很好地理解了内存管理和 C#提供的指针功能，也就能很好地集成 C#代码和原来的代码，并能在非常注重性能的系统中高效地处理内存。本章介绍了使用 C# 7 中的 ref 关键字作为返回类型和本地变量的新方法。这个特性减少了对使用不安全代码和 C#中指针的需要。本章还讨论了使用 Span 类型访问不同类型内存的更多细节，例如托管堆、本机堆和堆栈。

## 17.2　后台内存管理

C#编程的一个优点是程序员不需要担心具体的内存管理，垃圾收集器会自动处理所有的内存清理工作。用户可以得到像 C++语言那样的效率，而不需要考虑像在 C++中那样内存管理工作的复杂性。虽然不必手动管理内存，但仍需要理解后台发生的事情。理解程序在后台如何管理内存有助于提高应用程序的速度和性能。本节要介绍给变量分配内存时在计算机的内存中发生的情况。

**注意：**

*本节不详细介绍许多主题的相关内容。应把这一节看作是一般过程的简化向导，而不是实现的确切说明。*

### 17.2.1　值数据类型

Windows 使用一个虚拟寻址系统，该系统把程序可用的内存地址映射到硬件内存中的实际地址上，这些任务完全由 Windows 在后台管理。其实际结果是 32 位处理器上的每个进程都可以使用 4GB 的内存——无论计算机上实际有多少物理内存(在 64 位处理器上，这个数字会更大)。这个 4GB 的内存实际上包含了程序的所有部分，包括可执行代码、代码加载的所有 DLL，以及程序运行时使用的所有变量的内容。这个 4GB 的内存称为虚拟地址空间，或虚拟内存。为了方便起见，本章将它简称为内存。

**注意：**

*默认情况下，.NET Core 应用程序是作为可移植应用程序构建的。只要在系统上安装了.NET Core 运行库，可移植的应用程序就可以在 Windows 和 Linux 的 32 位和 64 位环境上运行。并不是所有的 API 都可以在所有平台上使用，尤其是在使用本机 API 时。为此，可以按照第 1 章的解释，给.NET Core 应用程序指定专门的平台。*

4GB 中的每个存储单元都是从 0 开始往上排序的。要访问存储在内存的某个空间中的一个值，就需要提供表示该存储单元的数字。在任何复杂的高级语言中，编译器负责把人们可以理解的变量名转换为处理器可以理解的内存地址。

在处理器的虚拟内存中，有一个区域称为栈。栈存储不是对象成员的值数据类型。另外，在调用一个方法时，也使用栈存储传递给方法的所有参数的副本。为了理解栈的工作原理，需要注意在 C#中的变量作用域。如果变量 a 在变量 b 之前进入作用域，b 就会首先超出作用域。考虑下面的代码：

```
{
  int a;
  // do something
  {
    int b;
    // do something else
  }
}
```

首先声明变量 a。接着在内部代码块中声明了 b。然后内部代码块终止，b 就超出作用域，最后 a 超出作用域。所以 b 的生存期完全包含在 a 的生存期中。在释放变量时，其顺序总是与给它们分配内存的顺序相反，这就是栈的工作方式。

还要注意，b 在另一个代码块中(通过另一对嵌套的花括号来定义)。因此，它包含在另一个作用域中。这称为块作用域或结构作用域。

我们不知道栈具体在地址空间的什么地方，这些信息在进行 C#开发时是不需要知道的。栈指针(操作系统维护的一个变量)表示栈中下一个空闲存储单元的地址。程序第一次开始运行时，栈指针指向为栈保留的内存块末尾。栈实际上是向下填充的，即从高内存地址向低内存地址填充。当数据入栈后，栈指针就会随之调整，以始终指向下一个空闲存储单元。这种情况如图 17-1 所示。在该图中，显示了栈指针 800000(十六进制的 0xC3500)，下一个空闲存储单元是地址 799999。

下面的代码会告诉编译器，需要一些存储空间以存储一个整数和一个双精度浮点数，这些存储单元分别称为 nRacingCars 和 engineSize。声明每个变量的代码行表示开始请求访问这个变量，闭合花括号标识这两个变量超出作用域的地方。

```
{
  int nRacingCars = 10;
  double engineSize = 3000.0;
  // do calculations;
}
```

假定使用如图 17-1 所示的栈。nRacingCars 变量进入作用域，赋值为 10，这个值放在存储单元 799996~799999 上，这 4 个字节就在栈指针所指空间的下面。有 4 个字节是因为存储 int 要使用 4 个字节。为了容纳该 int，应从栈指针对应的值中减去 4，所以它现在指向位置 799996，即下一个空闲单元(799995)。

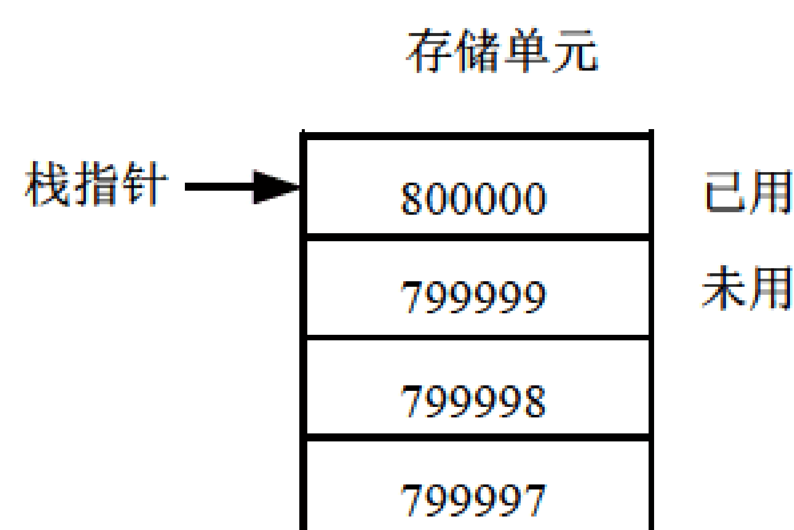



图 17-1

下一行代码声明变量 engineSize(这是一个 double 数)，把它初始化为 3000.0。一个 double 数要占用 8 个字节，所以值 3000.0 放在栈上的存储单元 799988~799995 上，栈指针对应的值减去 8，再次指向栈上的下一个空闲单元。

当 engineSize 超出作用域时，运行库就知道不再需要这个变量了。因为变量的生存期总是嵌套的，当 engineSize 在作用域中时，无论发生什么情况，都可以保证栈指针总是会指向存储 engineSize 的空间。为了从内存中删除这个变量，应给栈指针对应的值递增 8，现在它指向 engineSize 末尾紧接着的空间。此处就是放置闭合花括号的地方。当 nRacingCars 也超出作用域时，栈指针对应的值就再次递增 4。从栈中删除 engineSize 和 nRacingCars 之后，此时如果在作用域中又放入另一个变量，从 799999 开始的存储单元就会被覆盖，这些空间以前是存储 nRacingCars 的。

如果编译器遇到 int i，j 这样的代码行，则这两个变量进入作用域的顺序是不确定的。两个变量是同时声明的，也是同时超出作用域的。此时，变量以什么顺序从内存中删除就不重要了。编译器在内部会确保先放在内存中的那个变量后删除，这样就能保证该规则不会与变量的生存期冲突。

### 17.2.2 引用数据类型

尽管栈有非常高的性能，但它还没有灵活到可以用于所有的变量。变量的生存期必须嵌套，在许多情况下，这种要求都过于苛刻。通常我们希望使用一个方法分配内存，来存储一些数据，并在方法退出后的很长一段时

间内数据仍是可用的。只要是用 new 运算符来请求分配存储空间，就存在这种可能性——例如，对于所有的引用类型。此时就要使用托管堆。

如果读者以前编写过需要管理低级内存的 C++代码，就会很熟悉堆(heap)。托管堆和 C++使用的堆不同，它在垃圾收集器的控制下工作，与传统的堆相比有很显著的优势。

托管堆(简称为堆)是处理器的可用内存中的另一个内存区域。要了解堆的工作原理和如何为引用数据类型分配内存，看看下面的代码：

```
void DoWork()
{
  Customer arabel;
  arabel = new Customer();
  Customer otherCustomer2 = new EnhancedCustomer();
}
```

在这段代码中，假定存在两个类 Customer 和 EnhancedCustomer。EnhancedCustomer 类扩展了 Customer 类。

首先，声明一个 Customer 引用 arabel，在栈上给这个引用分配存储空间，但这仅是一个引用，而不是实际的 Customer 对象。arabel 引用占用 4 个字节的空间，足够包含 Customer 对象的存储地址(需要 4 个字节把 0～4GB 之间的内存地址表示为一个整数值)。

然后看下一行代码：

```
arabel = new Customer();
```

这行代码完成了以下操作：首先，它分配堆上的内存，以存储 Customer 对象(一个真正的对象，不只是一个地址)。然后把变量 arabel 的值设置为分配给新 Customer 对象的内存地址(它还调用合适的 Customer()构造函数初始化类实例中的字段，但此处我们不必担心这部分)。

Customer 实例没有放在栈中，而是放在堆中。在这个例子中，现在还不知道一个 Customer 对象占用多少字节，但为了讨论方便，假定是 32 个字节。这 32 个字节包含了 Customer 的实例字段，和.NET 用于识别和管理其类实例的一些信息。

为了在堆上找到存储新 Customer 对象的一个存储位置，.NET 运行库在堆中搜索，选取第一个未使用的且包含 32 个字节的连续块。为了讨论方便，假定其地址是 200000，arabel 引用占用栈中的 799996~799999 位置。这表示在实例化 arabel 对象前，内存的内容应如图 17-2 所示。

给 Customer 对象分配空间后，内存的内容应如图 17-3 所示。注意，与栈不同，堆上的内存是向上分配的，所以空闲空间在已用空间的上面。

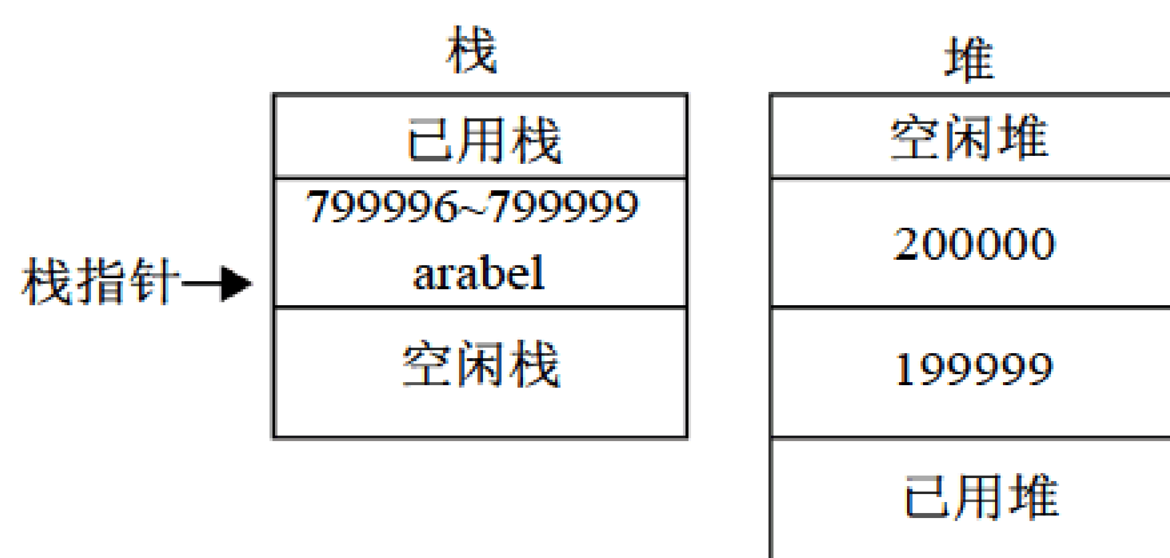



图 17-2

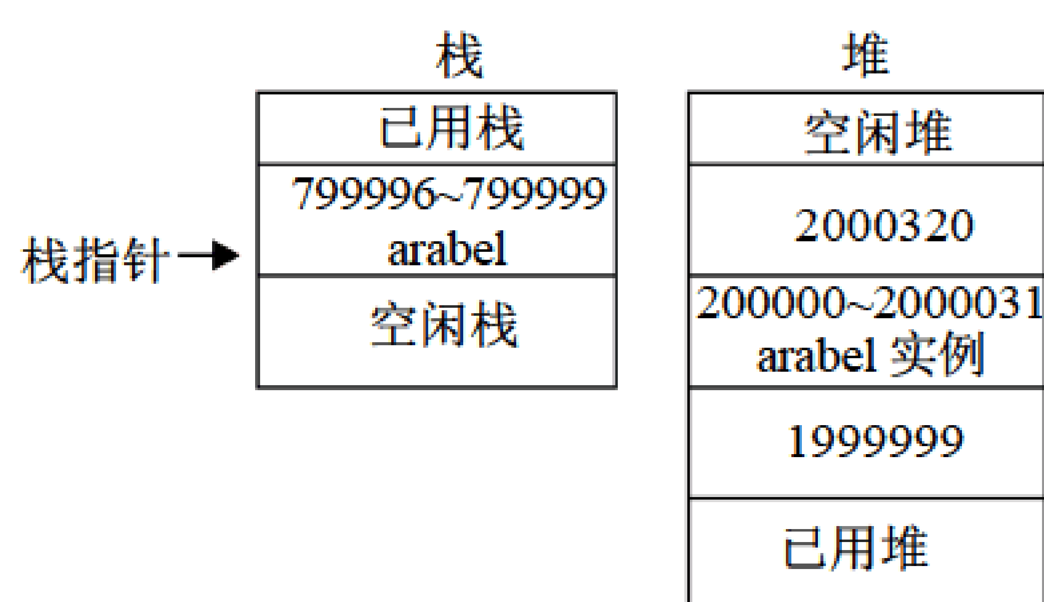



图 17-3

下一行代码声明了一个 Customer 引用，并实例化一个 Customer 对象。在这个例子中，用一行代码在栈上为 otherCustomer2 引用分配空间，同时在堆上为 mrJones 对象分配空间：

```
Customer otherCustomer2 = new EnhancedCustomer();
```

该行把栈上的 4 个字节分配给 otherCustomer2 引用，它存储在 799992~799995 位置上，而 otherCustomer2 对象在堆上从 200032 开始向上分配空间。

从这个例子可以看出，建立引用变量的过程要比建立值变量的过程更复杂，且不能避免性能的系统开销。实际上，我们对这个过程进行了过分的简化，因为.NET 运行库需要保存堆的状态信息，在堆中添加新数据时，这些信息也需要更新。尽管有这些性能开销，但仍有一种机制，在给变量分配内存时，不会受到栈的限制。把

一个引用变量的值赋予另一个相同类型的变量，就有两个变量引用内存中的同一对象了。当一个引用变量超出作用域时，它会从栈中删除，如上一节所述，但引用对象的数据仍保留在堆中，一直到程序终止，或垃圾收集器删除它为止，而只有在该数据不再被任何变量引用时，它才会被删除。

这就是引用数据类型的强大之处，在 C#代码中广泛使用了这个特性。这说明，我们可以对数据的生存期进行非常强大的控制，因为只要保持对数据的引用，该数据就肯定存在于堆上。

### 17.2.3 垃圾收集

由上面的讨论和图 17-3 和图 17-4 可以看出，托管堆的工作方式非常类似于栈，对象会在内存中一个挨一个地放置，这样就很容易使用指向下一个空闲存储单元的堆指针来确定下一个对象的位置。在堆上添加更多的对象时，也容易调整。但这比较复杂，因为基于堆的对象的生存期与引用它们的基于栈的变量的作用域不匹配。

在垃圾收集器运行时，它会从堆中删除不再引用的所有对象。垃圾收集器在引用的根表中找到所有引用的对象，接着在引用的对象树中查找。在完成删除操作后，堆会立即把对象分散开来，与已经释放的内存混合在一起，如图 17-4 所示。




图 17-4

如果托管的堆也是这样，在其上给新对象分配内存就成为一个很难处理的过程，运行库必须搜索整个堆，才能找到足够大的内存块来存储每个新对象。但是，垃圾收集器不会让堆处于这种状态。只要它释放了能释放的所有对象，就会把其他对象移动回堆的端部，再次形成一个连续的内存块。因此，堆可以继续像栈那样确定在什么地方存储新对象。当然，在移动对象时，这些对象的所有引用都需要用正确的新地址来更新，但垃圾收集器也会处理更新问题。

垃圾收集器的这个压缩操作是托管的堆与非托管的堆的区别所在。使用托管的堆，就只需要读取堆指针的值即可，而不需要遍历地址的链表，来查找一个地方放置新数据。

**注意：**

一般情况下，垃圾收集器在.NET 运行库确定需要进行垃圾收集时运行。可以调用 System.GC.Collect()方法，强迫垃圾收集器在代码的某个地方运行。System.GC 类是一个表示垃圾收集器的.NET 类，Collect()方法启动一个垃圾收集过程。但是，GC 类适用的场合很少，例如，代码中有大量的对象刚刚取消引用，就适合调用垃圾收集器。但是，垃圾收集器的逻辑不能保证在一次垃圾收集过程中，所有未引用的对象都从堆中删除。

**注意：**

在测试过程中运行 GC 是很有用的。这样，就可以看到应该收集的对象仍然未收集而导致的内存泄漏。因为垃圾收集器的工作做得很好，所以不要在生产代码中以编程方式收集内存。如果以编程方式调用 Collect，对象会更快地移入下一代，如下所示。这将导致 GC 运行更多的时间。

创建对象时，会把这些对象放在托管堆上。堆的第一部分称为第 0 代。创建新对象时，会把它们移动到堆

的这个部分中。因此，这里驻留了最新的对象。

在通过垃圾收集过程进行第一次对象收集之前，对象会继续放在这个部分。在此清理之后仍保留的对象会被压缩，然后移动到堆的下一个部分或堆的第 1 代对应的部分。

此时，第 0 代对应的部分为空，所有的新对象都再次放在这一部分上。在垃圾收集过程中遗留下来的旧对象放在第 1 代对应的部分上。老对象的这种移动会再次发生。接着重复下一次收集过程。这意味着，第 1 代中在垃圾收集过程中遗留下来的对象会移动到堆的第 2 代，位于第 0 代的对象会移动到第 1 代，第 0 代仍用于放置新对象。

**注意：**

*有趣的是，在给对象分配内存空间时，如果超出了第 0 代对应的部分的容量，或者调用了 GC.Collect()方法，就会进行垃圾收集。*

这个过程极大地提高了应用程序的性能。一般而言，最新的对象通常是可以收集的对象，而且可能也会收集大量比较新的对象。如果这些对象在堆中的位置是相邻的，垃圾收集过程就会更快。另外，相关的对象相邻放置也会使程序执行得更快。

在.NET 中，垃圾收集提高性能的另一个领域是架构处理堆上较大对象的方式。在.NET 下，较大对象有自己的托管堆，称为大对象堆。使用大于 85 000 个字节的对象时，它们就会放在这个特殊的堆上，而不是主堆上。.NET 应用程序不知道两者的区别，因为这是自动完成的。其原因是在堆上压缩大对象是比较昂贵的，因此驻留在大对象堆上的对象不执行压缩过程。

在进一步改进垃圾收集过程后，第二代和大对象堆上的收集现在放在后台线程上进行。这表示，应用程序线程仅会为第 0 代和第 1 代的收集而阻塞，减少了总暂停时间，对于大型服务器应用程序尤其如此。服务器和工作站默认打开这个功能。

有助于提高应用程序性能的另一个优化是垃圾收集的平衡，它专用于服务器的垃圾收集。服务器一般有一个线程池，执行大致相同的工作。内存分配在所有线程上都是类似的。对于服务器，每个逻辑服务器都有一个垃圾收集堆。因此其中一个堆用尽了内存，触发了垃圾收集过程时，所有其他堆也可能会得益于垃圾的收集。如果一个线程使用的内存远远多于其他线程，导致垃圾收集，其他线程可能不需要垃圾收集，这就不是很高效。垃圾收集过程会平衡这些堆——小对象堆和大对象堆。进行这个平衡过程，可以减少不必要的收集。

为了利用包含大量内存的硬件，垃圾收集过程添加了 GCSettings.LatencyMode 属性。把这个属性设置为 GCLatencyMode 枚举的一个值，可以控制垃圾收集器进行收集的方式。表 17-1 列出了 GCLatencyMode 可用的值。

表 17-1　GCLatencyMode 的设置

| 成　　员 | 说　　明 |
|---|---|
| Batch | 禁用并发设置，把垃圾收集设置为最大吞吐量。这会重写配置设置 |
| Interactive | 工作站的默认行为。它使用垃圾收集并发设置，平衡吞吐量和响应 |
| LowLatency | 保守的垃圾收集。只有系统存在内存压力时，才进行完整的收集。只应用于较短时间，执行特定的操作 |
| SustainedLowLatency | 只有系统存在内存压力时，才进行完整的内存块收集 |
| NoGCRegion | .NET 4.6 新增成员。对于 GCSettings，这是一个只读属性。可以在代码块中调用 GC.TryStartNoGC Region 和 EndNoGCRegion 来设置它。调用 TryStartNoGCRegion，定义需要可用的、GC 试图访问的内存大小。成功调用 TryStartNoGCRegion 后，指定不应运行的垃圾收集器，直到调用 EndNoGCRegion 为止 |

LowLatency 或 NoGCRegion 设置使用的时间应为最小值，分配的内存量应尽可能小。如果不小心，就可能出现溢出内存错误。

## 17.3 强引用和弱引用

垃圾收集器不能收集仍在引用的对象的内存——这是一个强引用。它可以收集不在根表中直接或间接引用的托管内存。然而，有时可能会忘记释放引用。

**注意：**
如果对象相互引用，但没有在根表中引用，例如，对象 A 引用 B，B 引用 C，C 引用 A，则 GC 可以销毁所有这些对象。

在应用程序代码内实例化一个类或结构时，只要有代码引用它，就会形成强引用。例如，如果有一个类 MyClass，并创建了一个变量 myClassVariable 来引用该类的对象，那么只要 myClassVariable 在作用域内，就存在对 MyClass 对象的强引用，如下所示：

```
var myClassVariable = new MyClass();
```

这意味着垃圾收集器不会清理 MyClass 对象使用的内存。一般而言这是好事，因为可能需要访问 MyClass 对象，可以创建一个缓存对象，它引用其他几个对象，如下：

```
var myCache = new MyCache();
myCache.Add(myClassVariable);
```

现在使用完 myClassVariable 了。它可以超出作用域，或指定为 null：

```
myClassVariable = null;
```

如果垃圾收集器现在运行，就不能释放 myClassVariable 引用的内存，因为该对象仍在缓存对象中引用。这样的引用可以很容易忘记，使用 WeakReference 可以避免这种情况。

弱引用允许创建和使用对象，但是垃圾收集器碰巧在运行，就会收集对象并释放内存。由于存在潜在的 bug 和性能问题，一般不会这么做，但是在特定的情况下使用弱引用是很合理的。弱引用对小对象也没有意义，因为弱引用有自己的开销，这个开销可能是比小对象更大。

弱引用是使用 WeakReference 类创建的。使用构造函数，可以传递强引用。示例代码创建了一个 DataObject，并传递构造函数返回的引用。在使用 WeakReference 时，可以检查 IsAlive 属性。再次使用该对象时，WeakReference 的 Target 属性就返回一个强引用。如果属性返回的值不是 null，就可以使用强引用。因为对象可能在任意时刻被收集，所以在引用该对象前必须确认它存在。成功检索强引用后，可以通过正常方式使用它，现在它不能被垃圾收集，因为它有一个强引用：

```
// Instantiate a weak reference to MathTest object
var myWeakReference = new WeakReference(new DataObject());
if (myWeakReference.IsAlive)
{
  DataObject strongReference = myWeakReference.Target as DataObject;
  if (strongReference != null)
  {
    // use the strongReference
  }
}
else
{
  // reference not available
}
```

## 17.4 处理非托管的资源

垃圾收集器的出现意味着，通常不需要担心不再需要的对象，只要让这些对象的所有引用都超出作用域，并允许垃圾收集器在需要时释放内存即可。但是，垃圾收集器不知道如何释放非托管的资源(例如，文件句柄、网络连接和数据库连接)。托管类在封装对非托管资源的直接或间接引用时，需要制定专门的规则，确保非托管的资源在收集类的一个实例时释放。

在定义一个类时，可以使用两种机制来自动释放非托管的资源。这些机制常常放在一起实现，因为每种机制都为问题提供了略为不同的解决方法。这两种机制是：

- 声明一个析构函数(或终结器)，作为类的一个成员
- 在类中实现 System.IDisposable 接口

下面依次讨论这两种机制，然后介绍如何同时实现它们，以获得最佳的效果。

### 17.4.1　析构函数或终结器

前面介绍了构造函数可以指定必须在创建类的实例时进行的某些操作。相反，在垃圾收集器销毁对象之前，也可以调用析构函数。由于执行这个操作，因此析构函数初看起来似乎是放置释放非托管资源、执行一般清理操作的代码的最佳地方。但是，事情并不是如此简单。

**注意：**

*在讨论 C#中的析构函数时，在底层的.NET 体系结构中，这些函数称为终结器(finalizer)。在 C#中定义析构函数时，编译器发送给程序集的实际上是 Finalize()方法。它不会影响源代码，但如果需要查看生成的 IL 代码，就应知道这个事实。*

C++开发人员应很熟悉析构函数的语法。它看起来类似于一个方法，与包含的类同名，但有一个前缀波形符(~)。它没有返回类型，不带参数，没有访问修饰符。下面是一个例子：

```
class MyClass
{
  ~MyClass()
  {
    // Finalizer implementation
  }
}
```

C#编译器在编译析构函数时，它会隐式地把析构函数的代码编译为等价于重写 Finalize()方法的代码，从而确保执行父类的 Finalize()方法。下面列出的 C#代码等价于编译器为~MyClass()析构函数生成的 IL：

```
protected override void Finalize()
{
  try
  {
    // Finalizer implementation
  }
  finally
  {
    base.Finalize();
  }
}
```

如上所示，在~MyClass()析构函数中实现的代码封装在 Finalize()方法的一个 try 块中。对父类的 Finalize()方法的调用放在 finally 块中，确保该调用的执行。第 14 章会讨论 try 块和 finally 块。

有经验的 C++开发人员大量使用了析构函数，有时不仅用于清理资源，还提供调试信息或执行其他任务。C#析构函数要比 C++析构函数的使用少得多。与 C++析构函数相比，C#析构函数的问题是它们的不确定性。在销毁 C++对象时，其析构函数会立即运行。但由于使用 C#时垃圾收集器的工作方式，无法确定 C#对象的析构函数何时执行。所以，不能在析构函数中放置需要在某一时刻运行的代码，也不应寄望于析构函数会以特定顺序对不同类的实例调用。如果对象占用了宝贵而重要的资源，应尽快释放这些资源，此时就不能等待垃圾收集器来释放了。

另一个问题是 C#析构函数的实现会延迟对象最终从内存中删除的时间。没有析构函数的对象会在垃圾收集器的一次处理中从内存中删除，但有析构函数的对象需要两次处理才能销毁：第一次调用析构函数时，没有删除对象，第二次调用才真正删除对象。另外，运行库使用一个线程来执行所有对象的 Finalize()方法。如果频繁使用析构函数，而且使用它们执行长时间的清理任务，对性能的影响就会非常显著。

### 17.4.2　IDisposable 接口

在 C#中，推荐使用 System.IDisposable 接口替代析构函数。IDisposable 接口定义了一种模式(具有语言级的支持)，该模式为释放非托管的资源提供了确定的机制，并避免产生析构函数固有的与垃圾收集器相关的问题。IDisposable 接口声明了一个 Dispose()方法，它不带参数，返回 void。MyClass 类的 Dispose()方法的实现代码如下：

```
class MyClass: IDisposable
{
  public void Dispose()
  {
    // implementation
  }
}
```

Dispose()方法的实现代码显式地释放由对象直接使用的所有非托管资源，并在所有也实现 IDisposable 接口的封装对象上调用 Dispose()方法。这样，Dispose()方法为何时释放非托管资源提供了精确的控制。

假定有一个 ResourceGobbler 类，它需要使用某些外部资源，且实现 IDisposable 接口。如果要实例化这个类的实例，使用它，然后释放它，就可以使用下面的代码：

```
var theInstance = new ResourceGobbler();
// do your processing
theInstance.Dispose();
```

但是，如果在处理过程中出现异常，这段代码就没有释放 theInstance 使用的资源，所以应使用 try 块，编写下面的代码：

```
ResourceGobbler theInstance = null;
try
{
  theInstance = new ResourceGobbler();
  // do your processing
}
finally
{
  theInstance?.Dispose();
}
```

### 17.4.3　using 语句

使用 try/finally，即使在处理过程中出现了异常，也可以确保总是在 theInstance 上调用 Dispose()方法，总是释放 theInstance 使用的任意资源。但是，如果总是要重复这样的结构，代码就很容易被混淆。C#提供了一种语法，可以确保在实现 IDisposable 接口的对象的引用超出作用域时，在该对象上自动调用 Dispose()方法。该语法使用了 using 关键字来完成此工作——该关键字在完全不同的环境下，它与名称空间没有关系。下面的代码生成与 try 块等价的 IL 代码：

```
using (var theInstance = new ResourceGobbler())
{
  // do your processing
}
```

using 语句的后面是一对圆括号，其中是引用变量的声明和实例化，该语句使变量的作用域限定在随后的语句块中。另外，在变量超出作用域时，即使出现异常，也会自动调用其 Dispose()方法。

**注意：**
using 关键字在 C#中有多个用法。using 声明用于导入名称空间。using 语句处理实现 IDisposable 的对象，并在作用域的末尾调用 Dispose 方法。

**注意：**
.NET Framework 中的几个类有 Close 和 Dispose 方法。如果常常要关闭资源(如文件和数据库)，就实现 Close

和 Dispose 方法。此时 Close()方法只是调用 Dispose()方法。这种方法在类的使用上比较清晰，还支持 using 语句。新类只实现了 Dispose 方法，因为我们已经习惯了它。

### 17.4.4　实现 IDisposable 接口和析构函数

前面的章节讨论了自定义类所使用的释放非托管资源的两种方式：

- 利用运行库强制执行的析构函数，但析构函数的执行是不确定的，而且，由于垃圾收集器的工作方式，它会给运行库增加不可接受的系统开销。
- IDisposable 接口提供了一种机制，该机制允许类的用户控制释放资源的时间，但需要确保调用 Dispose()方法。

如果创建了终结器，就应该实现 IDisposable 接口。假定大多数程序员都能正确调用 Dispose()方法，同时把实现析构函数作为一种安全机制，以防没有调用 Dispose()方法。下面是一个双重实现的例子：

```
public class ResourceHolder: IDisposable
{
  private bool _isDisposed = false;
  public void Dispose()
  {
    Dispose(true);
    GC.SuppressFinalize(this);
  }

  protected virtual void Dispose(bool disposing)
  {
    if (!_isDisposed)
    {
      if (disposing)
      {
        // Cleanup managed objects by calling their
        // Dispose() methods.
      }
      // Cleanup unmanaged objects
    }
    _isDisposed = true;
  }

  ~ResourceHolder()
  {
    Dispose(false);
  }

  public void SomeMethod()
  {
    // Ensure object not already disposed before execution of any method
    if(_isDisposed)
    {
      throw new ObjectDisposedException("ResourceHolder");
    }
    // method implementation...
  }
}
```

从上述代码可以看出，Dispose()方法有第二个 protected 重载方法，它带一个布尔参数，这是真正完成清理工作的方法。Dispose(bool)方法由析构函数和 IDisposable.Dispose()方法调用。这种方式的重点是确保所有的清理代码都放在一个地方。

传递给 Dispose(bool)方法的参数表示 Dispose(bool)方法是由析构函数调用，还是由 IDisposable.Dispose()方法调用——Dispose(bool)方法不应从代码的其他地方调用，其原因是：

- 如果使用者调用 IDisposable.Dispose()方法，该使用者就指定应清理所有与该对象相关的资源，包括托管和非托管的资源。
- 如果调用了析构函数，原则上所有的资源仍需要清理。但是在这种情况下，析构函数必须由垃圾收集器调用，而且用户不应试图访问其他托管的对象，因为我们不再能确定它们的状态了。在这种情况下，最好清理已知的非托管资源，希望任何引用的托管对象还有析构函数，这些析构函数执行自己的清理过程。

_isDisposed 成员变量表示对象是否已被清理，并确保不试图多次清理成员变量。它还允许在执行实例方法之前测试对象是否已清理，如 SomeMethod()方法所示。这个简单的方法不是线程安全的，需要调用者确保在同一时刻只有一个线程调用方法。要求使用者进行同步是一个合理的假定，在整个.NET 类库中(例如，在 Collection 类中)反复使用了这个假定。第 21 章将讨论线程和同步。

最后，IDisposable.Dispose()方法包含一个对 System.GC.SuppressFinalize()方法的调用。GC 类表示垃圾收集器，SuppressFinalize()方法则告诉垃圾收集器有一个类不再需要调用其析构函数了。因为 Dispose()方法已经完成了所有需要的清理工作，所以析构函数不需要做任何工作。调用 SuppressFinalize()方法就意味着垃圾收集器认为这个对象根本没有析构函数。

### 17.4.5 IDisposable 和终结器的规则

学习了终结器和 IDisposable 接口后，就已经了解了 Dispose 模式和使用这些构造的规则。因为释放资源是托管代码的一个重要方面，下面总结如下规则：

- 如果类定义了实现 IDisposable 的成员，该类也应该实现 IDisposable。
- 实现 IDisposable 并不意味着也应该实现一个终结器。终结器会带来额外的开销，因为它需要创建一个对象，释放该对象的内存，需要 GC 的额外处理。只在需要时才应该实现终结器，例如，发布本机资源。要释放本机资源，就需要终结器。
- 如果实现了终结器，也应该实现 IDisposable 接口。这样，本机资源可以早些释放，而不仅是在 GC 找出被占用的资源时，才释放资源。
- 在终结器的实现代码中，不能访问已终结的对象了。终结器的执行顺序是没有保证的。
- 如果所使用的一个对象实现了 IDisposable 接口，就在不再需要对象时调用 Dispose 方法。如果在方法中使用这个对象，using 语句比较方便。如果对象是类的一个成员，就让类也实现 IDisposable。

## 17.5 不安全的代码

如前所述，C#非常擅长于对开发人员隐藏大部分基本内存管理，因为它使用了垃圾收集器和引用。但是，有时需要直接访问内存。例如，由于性能问题，要在外部(非.NET 环境)的 DLL 中访问一个函数，该函数需要把一个指针当作参数来传递(许多 Windows API 函数就是这样)。本节将论述 C#直接访问内存的内容的功能。

### 17.5.1 用指针直接访问内存

下面把指针当作一个新论题来介绍，而实际上，指针并不是新东西。因为在代码中可以自由使用引用，而引用就是一个类型安全的指针。前面已经介绍了表示对象和数组的变量实际上存储相应数据(被引用者)的内存地址。指针只是一个以与引用相同的方式存储地址的变量。其区别是 C#不允许直接访问在引用变量中包含的地址。有了引用后，从语法上看，变量就可以存储引用的实际内容。

C#引用主要用于使 C#语言易于使用，防止用户无意中执行某些破坏内存中内容的操作。另一方面，使用指针，就可以访问实际的内存地址，执行新类型的操作。例如，给地址加上 4 个字节，就可以查看甚至修改存储在新地址中的数据。

下面是使用指针的两个主要原因：

- 向后兼容性——尽管.NET 运行库提供了许多工具，但仍可以调用本地的 Windows API 函数。对于某些操作，这可能是完成任务的唯一方式。这些 API 函数都是用 C++或 C#语言编写的，通常要求把指针作为其参数。但在许多情况下，还可以使用 DllImport 声明，以避免使用指针，例如，使用 System.IntPtr 类。
- 性能——在一些情况下，速度是最重要的，而指针可以提供最优性能。假定用户知道自己在做什么，就可以确保以最高效的方式访问或处理数据。但是，注意在代码的其他区域中，不使用指针，也可以对

性能进行必要的改进。请使用代码配置文件，查找代码中的瓶颈， Visual Studio 中就包含一个代码配置文件。

但是，这种低级的内存访问也是有代价的。使用指针的语法比引用类型的语法更复杂。而且，指针使用起来比较困难，需要非常高的编程技巧和很强的能力，仔细考虑代码所完成的逻辑操作，才能成功地使用指针。如果不仔细，使用指针就很容易在程序中引入细微的、难以查找的错误。例如，很容易重写其他变量，导致栈溢出，访问某些没有存储变量的内存区域，甚至重写.NET 运行库所需要的代码信息，因而使程序崩溃。

尽管有这些问题，但指针在编写高效的代码时是一种非常强大和灵活的工具。

**注意：**

这里强烈建议不要轻易使用指针，否则代码不仅难以编写和调试，而且无法通过 CLR 施加的内存类型安全检查。

### 1. 用 unsafe 关键字编写不安全的代码

因为使用指针会带来相关的风险，所以 C#只允许在特别标记的代码块中使用指针。标记代码所用的关键字是 unsafe。下面的代码把一个方法标记为 unsafe：

```
unsafe int GetSomeNumber()
{
  // code that can use pointers
}
```

任何方法都可以标记为 unsafe——无论该方法是否应用了其他修饰符(例如，静态方法、虚方法等)。在这种方法中，unsafe 修饰符还会应用到方法的参数上，允许把指针用作参数。还可以把整个类或结构标记为 unsafe，这表示假设所有的成员都是不安全的：

```
unsafe class MyClass
{
  // any method in this class can now use pointers
}
```

同样，可以把成员标记为 unsafe：

```
class MyClass
{
  unsafe int* pX; // declaration of a pointer field in a class
}
```

也可以把方法中的一块代码标记为 unsafe：

```
void MyMethod()
{
  // code that doesn't use pointers
  unsafe
  {
    // unsafe code that uses pointers here
  }

  // more 'safe' code that doesn't use pointers
}
```

但要注意，不能把局部变量本身标记为 unsafe：

```
int MyMethod()
{
  unsafe int *pX; // WRONG
}
```

如果要使用不安全的局部变量，就需要在不安全的方法或语句块中声明和使用它。在使用指针前还有一步要完成。C#编译器会拒绝不安全的代码，除非告诉编译器代码包含不安全的代码块。可以通过设置 csproj 项目文件的 AllowUnsafeBlocks，如图 17-5 所示，或者在 Visual Studio Build Project Properties 设置中选择 Allow Unsafe Code 复选框，配置不安全的代码：

```
<PropertyGroup>
  <AllowUnsafeBlocks>True</AllowUnsafeBlocks>
</PropertyGroup>
```

### 2. 指针的语法

把代码块标记为 unsafe 后，就可以使用下面的语法声明指针：

```
int* pWidth, pHeight;
double* pResult;
byte*[] pFlags;
```

这段代码声明了 4 个变量，pWidth 和 pHeight 是整数指针，pResult 是 double 型指针，pFlags 是字节型的数组指针。我们常常在指针变量名的前面使用前缀 p 来表示这些变量是指针。在变量声明中，符号*表示声明一个指针，换言之，就是存储特定类型的变量的地址。

声明了指针类型的变量后，就可以用与一般变量相同的方式使用它们，但首先需要学习另外两个运算符：

- &表示“取地址”，并把一个值数据类型转换为指针，例如，int 转换为*int。这个运算符称为寻址运算符。
- *表示“获取地址的内容”，把一个指针转换为值数据类型(例如，*float 转换为 float)。这个运算符称为“间接寻址运算符”(有时称为“取消引用运算符”)。

从这些定义中可以看出，&和*的作用是相反的。

> **注意：**
> 符号&和*也表示按位 AND(&)和乘法(*)运算符，为什么还可以以这种方式使用它们？答案是在实际使用时它们是不会混淆的，用户和编译器总是知道在什么情况下这两个符号有什么含义，因为按照指针的定义，这些符号总是以一元运算符的形式出现——它们只作用于一个变量，并出现在代码中该变量的前面。另一方面，按位 AND 和乘法运算符是二元运算符，它们需要两个操作数。

下面的代码说明了如何使用这些运算符：

```
int x = 10;
int* pX, pY;
pX = &x;
pY = pX;
*pY = 20;
```

首先声明一个整数 x，其值是 10。接着声明两个整数指针 pX 和 pY。然后把 pX 设置为指向 x(换言之，把 pX 的内容设置为 x 的地址)。然后把 pX 的值赋予 pY，所以 pY 也指向 x。最后，在语句*pY = 20 中，把值 20 赋予 pY 指向的地址包含的内容。实际上是把 x 的内容改为 20，因为 pY 指向 x。注意在这里，变量 pY 和 x 之间没有任何关系。只是此时 pY 碰巧指向存储 x 的存储单元而已。

要进一步理解这个过程，假定 x 存储在栈的存储单元 0x12F8C4～0x12F8C7 中(十进制就是 1243332～1243335，即有 4 个存储单元，因为一个 int 占用 4 个字节)。因为栈向下分配内存，所以变量 pX 存储在 0x12F8C0～0x12F8C3 的位置上，pY 存储在 0x12F8BC～0x12F8BF 的位置上。注意，pX 和 pY 也分别占用 4 个字节。这不是因为一个 int 占用 4 个字节，而是因为在 32 位处理器上，需要用 4 个字节存储一个地址。利用这些地址，在执行完上述代码后，栈应如图 17-5 所示。

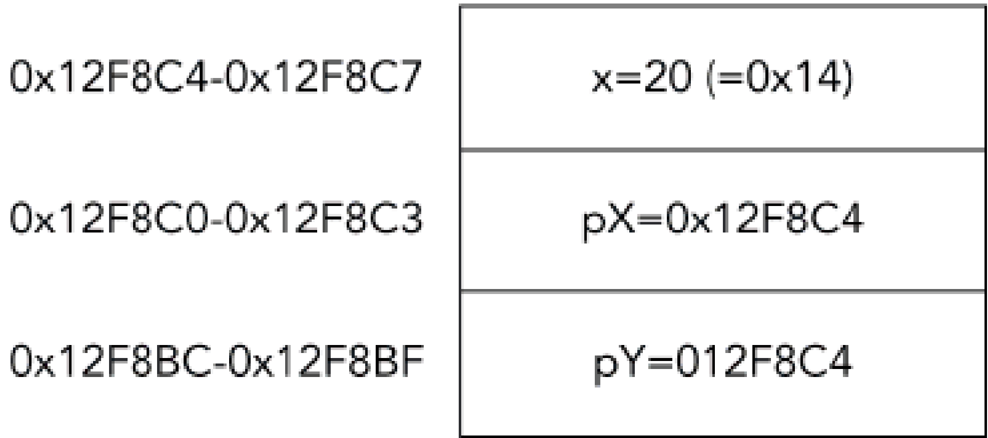



图 17-5

> **注意：**
> 这个示例使用 int 说明该过程，其中 int 存储在 32 位处理器中栈的连续空间上，但并不是所有的数据类型都会存储在连续的空间中。原因是 32 位处理器最擅长于在 4 个字节的内存块中检索数据。这种计算机上的内存会分解为 4 个字节的块，在 Windows 上，每个块有时称为 DWORD，因为这是 32 位无符号 int 数在.NET 出现之前的名字。这是从内存中获取 DWORD 的最高效的方式——跨越 DWORD 边界存储数据通常会降低硬件的性

能。因此，.NET 运行库通常会给某些数据类型填充一些空间，使它们占用的内存是 4 的倍数。例如，short 数据占用两个字节，但如果把一个 short 放在栈中，栈指针仍会向下移动 4 个字节，而不是两个字节，这样，下一个存储在栈中的变量就仍从 DWORD 的边界开始存储。

可以把指针声明为任意一种值类型——即任何预定义的类型 uint、int 和 byte 等，也可以声明为一个结构。但是不能把指针声明为一个类或数组，因为这么做会使垃圾收集器出现问题。为了正常工作，垃圾收集器需要知道在堆上创建了什么类的实例，它们在什么地方。但如果代码开始使用指针处理类，就很容易破坏堆中.NET 运行库为垃圾收集器维护的与类相关的信息。在这里，垃圾收集器可以访问的任何数据类型称为托管类型，而指针只能声明为非托管类型，因为垃圾收集器不能处理它们。

### 3. 将指针强制转换为整数类型

由于指针实际上存储了一个表示地址的整数，因此任何指针中的地址都可以和任何整数类型之间相互转换。指针到整数类型的转换必须是显式指定的，隐式的转换是不允许的。例如，编写下面的代码是合法的：

```
int x = 10;
int* pX, pY;
pX = &x;
pY = pX;
*pY = 20;
ulong y = (ulong)pX;
int* pD = (int*)y;
```

把指针 pX 中包含的地址强制转换为一个 uint，存储在变量 y 中。接着把 y 强制转换回一个 int*，存储在新变量 pD 中。因此 pD 也指向 x 的值。

把指针的值强制转换为整数类型的主要目的是显示它。虽然插入字符串和 Console.Write()方法没有带指针的重载方法，但是必须把指针的值强制转换为整数类型，这两个方法才能接受和显示它们：

```
WriteLine($"Address is {pX}"); // wrong -- will give a compilation error
WriteLine($"Address is {(ulong)pX}"); // OK
```

可以把一个指针强制转换为任何整数类型，但是，因为在 32 位系统上，一个地址占用 4 个字节，把指针强制转换为除了 uint、long 或 ulong 之外的数据类型，肯定会导致溢出错误(int 也可能导致这个问题，因为它的取值范围是–20 亿~20 亿，而地址的取值范围是 0~40 亿)。如果创建 64 位应用程序，就需要把指针强制转换为 ulong 类型。

还要注意，checked 关键字不能用于涉及指针的转换。对于这种转换，即使在设置 checked 的情况下，发生溢出时也不会抛出异常。.NET 运行库假定，如果使用指针，就知道自己要做什么，不必担心可能出现的溢出。

### 4. 指针类型之间的强制转换

也可以在指向不同类型的指针之间进行显式的转换。例如：

```
byte aByte = 8;
byte* pByte= &aByte;
double* pDouble = (double*)pByte;
```

这是一段合法的代码，但如果要执行这段代码，就要小心了。在上面的示例中，如果要查找指针 pDouble 指向的 double 值，就会查找包含 1 个 byte(aByte)的内存，和一些其他内存，并把它当作包含一个 double 值的内存区域来对待——这不会得到一个有意义的值。但是，可以在类型之间转换，实现 C union 类型的等价形式，或者把指针强制转换为其他类型，例如，把指针转换为 sbyte，来检查内存的单个字节。

### 5. void 指针

如果要维护一个指针，但不希望指定它指向的数据类型，就可以把指针声明为 void：

```
int* pointerToInt;
void* pointerToVoid;
pointerToVoid = (void*)pointerToInt;
```

void 指针的主要用途是调用需要 void*参数的 API 函数。在 C#语言中，使用 void 指针的情况并不是很多。

特殊情况下，如果试图使用*运算符取消引用 void 指针，编译器就会标记一个错误。

#### 6. 指针算术的运算

可以给指针加减整数。但是，编译器很智能，知道如何执行这个操作。例如，假定有一个 int 指针，要在其值上加 1。编译器会假定我们要查找 int 后面的存储单元，因此会给该值加上 4 个字节，即加上一个 int 占用的字节数。如果这是一个 double 指针，加 1 就表示在指针的值上加 8 个字节，即一个 double 占用的字节数。只有指针指向 byte 或 sbyte(都是 1 个字节)时，才会给该指针的值加上 1。

可以对指针使用运算符+、–、+=、–=、++和––，这些运算符右边的变量必须是 long 或 ulong 类型。

> **注意：**
> 不允许对 void 指针执行算术运算。

例如，假定有如下定义：

```
uint u = 3;
byte b = 8;
double d = 10.0;
uint* pUint= &u; // size of a uint is 4
byte* pByte = &b; // size of a byte is 1
double* pDouble = &d; // size of a double is 8
```

下面假定这些指针指向的地址是：

- pUint: 1243332
- pByte: 1243328
- pDouble: 1243320

执行这段代码后：

```
++pUint; // adds (1*4) = 4 bytes to pUint
pByte -= 3; // subtracts (3*1) = 3 bytes from pByte
double* pDouble2 = pDouble + 4; // pDouble2 = pDouble + 32 bytes (4*8 bytes)
```

指针应包含的内容是：

- pUint: 1243336
- pByte: 1243325
- pDouble2: 1243352

> **注意：**
> 一般规则是，给类型为 T 的指针加上数值 X，其中指针的值为 P，则得到的结果是 P+ X*(sizeof(T))。使用这条规则时要小心。如果给定类型的连续值存储在连续的存储单元中，指针加法就允许在存储单元之间移动指针。但如果类型是 byte 或 char，其总字节数不是 4 的倍数，连续值就不是默认地存储在连续的存储单元中。

如果两个指针都指向相同的数据类型，则也可以把一个指针从另一个指针中减去。此时，结果是一个 long，其值是指针值的差被该数据类型所占用的字节数整除的结果：

```
double* pD1 = (double*)1243324; // note that it is perfectly valid to
// initialize a pointer like this.
double* pD2 = (double*)1243300;
long L = pD1-pD2; // gives the result 3 (=24/sizeof(double))
```

#### 7. sizeof 运算符

这一节将介绍如何确定各种数据类型的大小。如果需要在代码中使用某种类型的大小，就可以使用 sizeof 运算符，它的参数是数据类型的名称，返回该类型占用的字节数。例如：

```
int x = sizeof(double);
```

这将设置 x 的值为 8。

使用 sizeof 的优点是不必在代码中硬编码数据类型的大小，使代码的移植性更强。对于预定义的数据类型，

sizeof 返回下面的值。

```
sizeof(sbyte) = 1; sizeof(byte) = 1;
sizeof(short) = 2; sizeof(ushort) = 2;
sizeof(int) = 4; sizeof(uint) = 4;
sizeof(long) = 8; sizeof(ulong) = 8;
sizeof(char) = 2; sizeof(float) = 4;
sizeof(double) = 8; sizeof(bool) = 1;
```

也可以对自己定义的结构使用 sizeof，但此时得到的结果取决于结构中的字段类型。不能对类使用 sizeof。

#### 8. 结构指针：指针成员访问运算符

结构指针的工作方式与预定义值类型的指针的工作方式完全相同。但是这有一个条件：结构不能包含任何引用类型，这是因为前面介绍的一个限制——指针不能指向任何引用类型。为了避免这种情况，如果创建一个指针，它指向包含任何引用类型的任何结构，编译器就会标记一个错误。

假定定义了如下结构：

```
struct MyStruct
{
  public long X;
  public float F;
}
```

就可以给它定义一个指针：

```
MyStruct* pStruct;
```

然后对其进行初始化：

```
var myStruct = new MyStruct();
pStruct = &myStruct;
```

也可以通过指针访问结构的成员值：

```
(*pStruct).X = 4;
(*pStruct).F = 3.4f;
```

但是，这个语法有点复杂。因此，C#定义了另一个运算符，用一种比较简单的语法，通过指针访问结构的成员，它称为指针成员访问运算符，其符号是一个短划线，后跟一个大于号，它看起来像一个箭头：->。

**注意：**
C++开发人员能识别指针成员访问运算符。因为 C++使用这个符号完成相同的任务。

使用这个指针成员访问运算符，上述代码可以重写为：

```
pStruct->X = 4;
pStruct->F = 3.4f;
```

也可以直接把合适类型的指针设置为指向结构中的一个字段：

```
long* pL = &(Struct.X);
float* pF = &(Struct.F);
```

或者

```
long* pL = &(pStruct->X);
float* pF = &(pStruct->F);
```

#### 9. 类成员的指针

前面说过，不能创建指向类的指针，这是因为垃圾收集器不维护关于指针的任何信息，只维护关于引用的信息，因此创建指向类的指针会使垃圾收集器不能正常工作。

但是，大多数类都包含值类型的成员，可以为这些值类型成员创建指针，但这需要一种特殊的语法。例如，假定把上面示例中的结构重写为类：

```
class MyClass
{
  public long X;
  public float F;
}
```

然后就可以为它的字段 X 和 F 创建指针了，方法与前面一样。但这么做会产生一个编译错误：

```
var myObject = new MyClass();
long* pL = &(myObject.X); // wrong -- compilation error
float* pF = &(myObject.F); // wrong -- compilation error
```

尽管 X 和 F 都是非托管类型，但它们嵌入在一个对象中，这个对象存储在堆上。在垃圾收集的过程中，垃圾收集器会把 MyObject 移动到内存的一个新单元上，这样，pL 和 pF 就会指向错误的存储地址。由于存在这个问题，因此编译器不允许以这种方式把托管类型的成员的地址分配给指针。

解决这个问题的方法是使用 fixed 关键字，它会告诉垃圾收集器，可能有引用某些对象的成员的指针，所以这些对象不能移动。如果要声明一个指针，则使用 fixed 的语法，如下所示：

```
var myObject = new MyClass();
fixed (long* pObject = &(myObject.X))
{
  // do something
}
```

在关键字 fixed 后面的圆括号中，定义和初始化指针变量。这个指针变量(在本例中是 pObject)的作用域是花括号标识的 fixed 块。这样，垃圾收集器就知道，在执行 fixed 块中的代码时，不能移动 myObject 对象。

如果要声明多个这样的指针，就可以在同一个代码块前放置多条 fixed 语句：

```
var myObject = new MyClass();
fixed (long* pX = &(myObject.X))
fixed (float* pF = &(myObject.F))
{
  // do something
}
```

如果要在不同的阶段固定几个指针，就可以嵌套整个 fixed 块：

```
var myObject = new MyClass();
fixed (long* pX = &(myObject.X))
{
  // do something with pX
  fixed (float* pF = &(myObject.F))
  {
    // do something else with pF
  }
}
```

如果这些变量的类型相同，就可以在同一个 fixed 块中初始化多个变量：

```
var myObject = new MyClass();
var myObject2 = new MyClass();
fixed (long* pX = &(myObject.X), pX2 = &(myObject2.X))
{
  // etc.
}
```

在上述情况中，是否声明不同的指针，让它们指向相同或不同对象中的字段，或者指向与类实例无关的静态字段，这一点并不重要。

## 17.5.2 指针示例：PointerPlayground

为了理解指针，最好编写一个使用指针的程序，再使用调试器。下面给出一个使用指针的示例：PointerPlayground。它执行一些简单的指针操作，显示结果，还允许查看内存中发生的情况，并确定变量存储在什么地方(代码文件 PointerPlayground/Program.cs)：

```
class Program
{
  unsafe static void Main()
  {
    int x=10;
    short y = -1;
    byte y2 = 4;
    double z = 1.5;
    int* pX = &x;
    short* pY = &y;
    double* pZ = &z;
```

```
      Console.WriteLine($"Address of x is 0x{(ulong)&x:X}, " +
        $"size is {sizeof(int)}, value is {x}");
      Console.WriteLine($"Address of y is 0x{(ulong)&y2:X}, " +
        $"size is {sizeof(short)}, value is {y}");
      Console.WriteLine($"Address of y2 is 0x{(ulong)&y2:X}, " +
        $"size is {sizeof(byte)}, value is {y2}");
      Console.WriteLine($"Address of z is 0x{(ulong)&z:X}, " +
        $"size is {sizeof(double)}, value is {z}");
      Console.WriteLine($"Address of pX=&x is 0x{(ulong)&pX:X}, " +
        $"size is {sizeof(int*)}, value is 0x{(ulong)pX:X}");
      Console.WriteLine($"Address of pY=&y is 0x{(ulong)&pY:X}, " +
        $"size is {sizeof(short*)}, value is 0x{(ulong)pY:X}");
      Console.WriteLine($"Address of pZ=&z is 0x{(ulong)&pZ:X}, " +
        $"size is {sizeof(double*)}, value is 0x{(ulong)pZ:X}");
      *pX = 20;
      Console.WriteLine($"After setting *pX, x = {x}");
      Console.WriteLine($"*pX = {*pX}");
      pZ = (double*)pX;
      Console.WriteLine($"x treated as a double = {*pZ}");
      Console.ReadLine();
    }
  }
```

这段代码声明了 4 个值变量：

- int x
- short y
- byte y2
- double z

它还声明了指向其中 3 个值的指针：pX、pY 和 pZ。

然后显示这 3 个变量的值，以及它们的大小和地址。注意在获取 pX、pY 和 pZ 的地址时，我们查看的是指针的指针，即值的地址的地址！还要注意，与显示地址的常见方式一致，在 WriteLine()命令中使用{0:X}格式说明符，确保该内存地址以十六进制格式显示。

最后，使用指针 pX 把 x 的值改为 20，执行一些指针类型强制转换，如果把 x 的内容当作 double 类型，就会得到无意义的结果。

编译并运行这段代码，得到下面的结果：

```
Address of x is 0x376943D5A8, size is 4, value is 10
Address of y is 0x376943D5A0, size is 2, value is -1
Address of y2 is 0x376943D598, size is 1, value is 4
Address of z is 0x376943D590, size is 8, value is 1.5
Address of pX=&x is 0x376943D588, size is 8, value is 0x376943D5A8
Address of pY=&y is 0x376943D580, size is 8, value is 0x376943D5A0
Address of pZ=&z is 0x376943D578, size is 8, value is 0x376943D590
After setting *pX, x = 20
*pX = 20
x treated as a double = 9.88131291682493E-323
```

**注意：**

*用 CoreCLR 运行应用程序时，每次运行应用程序都会显示不同的地址。*

检查这些结果，可以证实“后台内存管理”一节描述的栈操作，即栈向下给变量分配内存。注意，这还证实了栈中的内存块总是按照 4 个字节的倍数进行分配。例如，y 是一个 short 数(其大小为 2 字节)，其地址是 0xD4E710(十六进制)，表示为该变量分配的存储单元是 0xD4E710~0xD4E713。如果.NET 运行库严格地逐个排列变量，则 y 应只占用两个存储单元，即 0xD4E712 和 0xD4E713。

下一个示例 PointerPlayground2 介绍指针的算术，以及结构指针和类成员。开始时，定义一个结构 CurrencyStruct，它把货币值表示为美元和美分，再定义一个等价的类 CurrencyClass(代码文件 PointerPlayground2/Currency.cs)：

```
internal struct CurrencyStruct
{
  public long Dollars;
  public byte Cents;
```

```
  public override string ToString() => $"$ {Dollars}.{Cents}";
}

internal class CurrencyClass
{
  public long Dollars = 0;
  public byte Cents = 0;
  public override string ToString() => $"$ {Dollars}.{Cents}";
}
```

定义好结构和类后，就可以对它们应用指针了。下面的代码是一个新的示例。这段代码比较长，我们对此将做详细讲解。首先显示 CurrencyStruct 结构的字节数，创建它的两个实例和一些指针，然后使用 pAmount 指针初始化一个 CurrencyStruct 结构 amount1 的成员，显示变量的地址(代码文件 PointerPlayground2/Program.cs )：

```
unsafe static void Main()
{
  Console.WriteLine($"Size of CurrencyStruct struct is " +
    $"{sizeof(CurrencyStruct)}");
  CurrencyStruct amount1, amount2;
  CurrencyStruct* pAmount = &amount1;
  long* pDollars = &(pAmount->Dollars);
  byte* pCents = &(pAmount->Cents);
  Console.WriteLine("Address of amount1 is 0x{(ulong)&amount1:X}");
  Console.WriteLine("Address of amount2 is 0x{(ulong)&amount2:X}");
  Console.WriteLine("Address of pAmount is 0x{(ulong)&pAmount:X}");
  Console.WriteLine("Address of pDollars is 0x{(ulong)&pDollars:X}");
  Console.WriteLine("Address of pCents is 0x{(ulong)&pCents:X}");

  pAmount->Dollars = 20;
  *pCents = 50;
  Console.WriteLine($"amount1 contains {amount1}");
  //...
}
```

现在根据栈的工作方式，执行一些指针操作。因为变量是按顺序声明的，所以 amount2 存储在 amount1 后面的地址中。sizeof(CurrencyStruct)运算符返回 16(见后面的屏幕输出)，所以 CurrencyStruct 结构占用的字节数是 4 的倍数。在递减了 Currency 指针后，它就指向 amount2：

```
--pAmount; // this should get it to point to amount2
Console.WriteLine($"amount2 has address 0x{(ulong)pAmount:X} " +
  $"and contains {*pAmount}");
```

在调用 Console.WriteLine()语句时，它显示了 amount2 的内容，但还没有对它进行初始化。显示出来的东西就是随机的垃圾——在执行该示例前内存中存储在该单元中的内容。但这有一个要点：一般情况下，C#编译器会禁止使用未初始化的变量，但在开始使用指针时，就很容易绕过许多通常的编译检查。此时我们这么做，是因为编译器无法知道我们实际上要显示的是 amount2 的内容。因为知道了栈的工作方式，所以可以说出递减 pAmount 的结果是什么。使用指针算术，可以访问编译器通常禁止访问的各种变量和存储单元，因此指针算术是不安全的。

接下来在 pCents 指针上进行指针运算。pCents 指针目前指向 amount1.Cents，但此处的目的是使用指针算术让它指向 amount2.Cents，而不是直接告诉编译器我们要做什么。为此，需要从 pCents 指针所包含的地址中减去 sizeof(Currency)：

```
// do some clever casting to get pCents to point to cents
// inside amount2
CurrencyStruct* pTempCurrency = (CurrencyStruct*)pCents;
pCents = (byte*) ( --pTempCurrency );
Console.WriteLine("Address of pCents is now 0x{0:X}", (ulong)&pCents);
```

最后，使用 fixed 关键字创建一些指向类实例中字段的指针，使用这些指针设置这个实例的值。注意，这也是我们第一次查看存储在堆中(而不是栈)的项的地址：

```
Console.WriteLine("\nNow with classes");
// now try it out with classes
var amount3 = new CurrencyClass();
fixed(long* pDollars2 = &(amount3.Dollars))
fixed(byte* pCents2 = &(amount3.Cents))
{
  Console.WriteLine($"amount3.Dollars has address 0x{(ulong)pDollars2:X}");
  Console.WriteLine($"amount3.Cents has address 0x{(ulong)pCents2:X}");
  *pDollars2 = -100;
```

```
    Console.WriteLine($"amount3 contains {amount3}");
  }
```

编译并运行这段代码，得到如下所示的结果：

```
Size of CurrencyStruct struct is 16
Address of amount1 is 0xD290DCD7C0
Address of amount2 is 0xD290DCD7B0
Address of pAmount is 0xD290DCD7A8
Address of pDollars is 0xD290DCD7A0
Address of pCents is 0xD290DCD798
amount1 contains $ 20.50
amount2 has address 0xD290DCD7B0 and contains $ 0.0
Address of pCents is now 0xD290DCD798
Now with classes
amount3.Dollars has address 0xD292C91A70
amount3.Cents has address 0xD292C91A78
amount3 contains $ -100.0
```

注意，在这个结果中，显示了未初始化的 amount2 的值，CurrencyStruct 结构的字节数是 16，大于其字段的字节数(一个 long 数占用 8 个字节，加上 1 个字节等于 9 个字节)。

### 17.5.3 使用指针优化性能

前面用许多篇幅介绍了使用指针可以完成的各种任务，但在前面的示例中，仅是处理内存，让有兴趣的人们了解实际上发生了什么事，并没有帮助人们编写出更好的代码！本节将应用我们对指针的理解，用一个示例来说明使用指针可以大大提高性能。

#### 1. 创建基于栈的数组

本节将探讨指针的一个主要应用领域：在栈中创建高性能、低系统开销的数组。第 2 章介绍了 C#如何支持数组的处理。第 7 章详细介绍了数组。C#很容易使用一维数组和矩形或锯齿形多维数组，但有一个缺点：这些数组实际上都是对象，它们是 System.Array 的实例。因此数组存储在堆上，这会增加系统开销。有时，我们希望创建一个使用时间比较短的高性能数组，不希望有引用对象的系统开销。而使用指针就可以做到，但指针只对于一维数组比较简单。

为了创建一个高性能的数组，需要使用另一个关键字：stackalloc。stackalloc 命令指示.NET 运行库在栈上分配一定量的内存。在调用 stackalloc 命令时，需要为它提供两条信息：

- 要存储的数据类型
- 需要存储的数据项数

例如，要分配足够的内存，以存储 10 个 decimal 数据项，可以编写下面的代码：

```
decimal* pDecimals = stackalloc decimal[10];
```

注意，这条命令只分配栈内存。它不会试图把内存初始化为任何默认值，这正好符合我们的目的。因为要创建一个高性能的数组，给它不必要地初始化相应值会降低性能。

同样，要存储 20 个 double 数据项，可以编写下面的代码：

```
double* pDoubles = stackalloc double[20];
```

虽然这行代码指定把变量的个数存储为一个常数，但它等于在运行时计算的一个数字。所以可以把上面的示例写为：

```
int size;
size = 20; // or some other value calculated at runtime
double* pDoubles = stackalloc double[size];
```

从这些代码段中可以看出，stackalloc 的语法有点不寻常。它的后面紧跟要存储的数据类型名(该数据类型必须是一个值类型)，之后把需要的项数放在方括号中。分配的字节数是项数乘以 sizeof(数据类型)。在这里，使用方括号表示这是一个数组。如果给 20 个 double 数分配存储单元，就得到了一个有 20 个元素的 double 数组，最简单的数组类型是逐个存储元素的内存块，如图 17-6 所示。

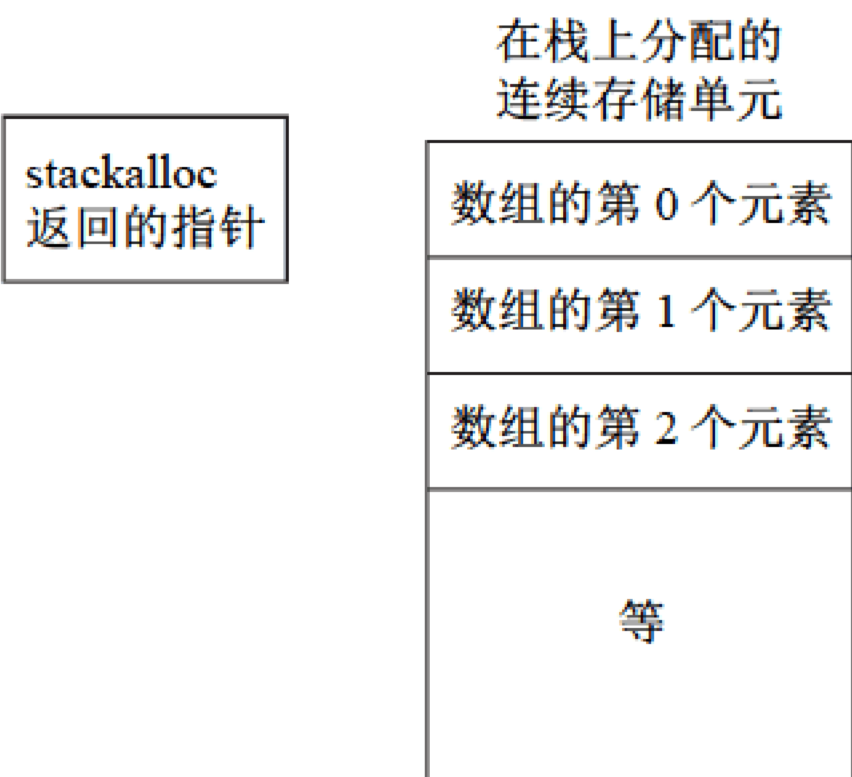



图 17-6

在图 17-6 中，显示了 stackalloc 返回的指针，stackalloc 总是返回分配数据类型的指针，它指向新分配内存块的顶部。要使用这个内存块，可以取消对已返回指针的引用。例如，给 20 个 double 数分配内存后，把第一个元素(数组的元素 0)设置为 3.0，可以编写下面的代码：

```
double* pDoubles = stackalloc double[20];
*pDoubles = 3.0;
```

要访问数组的下一个元素，可以使用指针算术。如前所述，如果给一个指针加 1，它的值就会增加它指向的数据类型的字节数。在本例中，就会把指针指向已分配的内存块中的下一个空闲存储单元。因此可以把数组的第二个元素(元素编号为 1)设置为 8.4：

```
double* pDoubles = stackalloc double[20];
*pDoubles = 3.0;
*(pDoubles + 1) = 8.4;
```

同样，可以用表达式*(pDoubles+X)访问数组中下标为 X 的元素。

这样，就得到一种访问数组中元素的方式，但对于一般目的，使用这种语法过于复杂。C#为此定义了另一种语法。对指针应用方括号时，C#为方括号提供了一种非常精确的含义。如果变量 p 是任意指针类型，X 是一个整数，表达式 p[X]就被编译器解释为*(p+X)，这适用于所有的指针，不仅仅是用 stackalloc 初始化的指针。利用这个简洁的表示法，就可以用一种非常方便的语法访问数组。实际上，访问基于栈的一维数组所使用的语法与访问由 System.Array 类表示的基于堆的数组完全相同：

```
double* pDoubles = stackalloc double [20];
pDoubles[0] = 3.0; // pDoubles[0] is the same as *pDoubles
pDoubles[1] = 8.4; // pDoubles[1] is the same as *(pDoubles+1)
```

**注意：**

把数组的语法应用于指针并不是新东西。自从开发出 C 和 C++语言以来，它就是这两种语言的基础部分。实际上，C++开发人员会把这里用 stackalloc 获得的、基于栈的数组完全等同于传统的基于栈的 C 和 C++数组。这种语法和指针与数组的链接方式是 C 语言在 20 世纪 70 年代后期流行起来的原因之一，也是指针的使用成为 C 和 C++中一种流行的编程技巧的主要原因。

尽管高性能的数组可以用与一般 C#数组相同的方式访问，但需要注意：在 C#中，下面的代码会抛出一个异常：

```
double[] myDoubleArray = new double [20];
myDoubleArray[50] = 3.0;
```

抛出异常的原因是：使用越界的下标来访问数组；下标是 50，而允许的最大下标是 19。但是，如果使用 stackalloc 声明了一个等价的数组，对数组进行边界检查时，这个数组中就没有封装任何对象，因此下面的代码不会抛出异常：

```
double* pDoubles = stackalloc double [20];
pDoubles[50] = 3.0;
```

在这段代码中，我们分配了足够的内存来存储 20 个 double 类型的数。接着把 sizeof(double)存储单元的起

始位置设置为该存储单元的起始位置加上 50*sizeof(double)个存储单元，来保存双精度值 3.0。但这个存储单元超出了刚才为 double 数分配的内存区域。谁也不知道这个地址存储了什么数据。最好是只使用某个当前未使用的内存，但所重写的存储单元也有可能是在栈上用于存储其他变量，或者是某个正在执行的方法的返回地址。因此，使用指针获得高性能的同时，也会付出一些代价：需要确保自己知道在做什么，否则就会抛出非常古怪的运行错误。

### 2. QuickArray 示例

下面用一个 stackalloc 示例 QuickArray 来结束关于指针的讨论。在这个示例中，程序仅要求用户提供为数组分配的元素数。然后代码使用 stackalloc 给 long 型数组分配一定的存储单元。这个数组的元素是从 0 开始的整数的平方，结果显示在控制台上(代码文件 QuickArray/Program.cs)：

```
class Program
{
  unsafe static void Main()
  {
    Console.Write("How big an array do you want? \n> ");
    string userInput = ReadLine();
    uint size = uint.Parse(userInput);
    long* pArray = stackalloc long[(int) size];
    for (int i = 0; i < size; i++)
    {
      pArray[i] = i*i;
    }

    for (int i = 0; i < size; i++)
    {
      Console.WriteLine($"Element {i} = {*(pArray + i)}");
    }
    Console.ReadLine();
  }
}
```

运行这个示例，得到如下所示的结果：

```
How big an array do you want?
> 15
Element 0 = 0
Element 1 = 1
Element 2 = 4
Element 3 = 9
Element 4 = 16
Element 5 = 25
Element 6 = 36
Element 7 = 49
Element 8 = 64
Element 9 = 81
Element 10 = 100
Element 11 = 121
Element 12 = 144
Element 13 = 169
Element 14 = 196
_
```

## 17.6　引用的语义

第 3 章展示了在将参数传递给方法时所使用的 ref 关键字。当通过值传递结构时，将复制结构的内容。通过引用传递结构(使用 ref 关键字)，新变量会引用相同的数据。

通过 C# 7.0，还可以使用 ref 关键字作为返回类型的修饰符，并作为本地变量的修饰符。在 C# 7.2 中，可以将 readonly 修饰符添加到 ref 关键字，以不允许更改。C# 7.2 还添加了 in 关键字，以通过引用传递值类型，而不允许它们发生更改。本节将讨论这些新特性。

一方面，最好有不可变类型，因为这些类型允许从多个线程中访问，而不需要同步，因为没有线程可以更改值。然而，不可变类型也意味着需要复制大量数据。对于值类型，需要复制数据，当然，这也会降低性能。使用引用类型，需要不同的变量来引用堆上的相同数据，而且这些数据可能也需要一个副本。例如，string 类

型是不可变的。像 ToUpper 和 ToLower 这样 string 类型的方法永远不会更改字符串，而是返回一个新字符串。当这些对象不再被引用时，需要收集垃圾。为了避免过度使用垃圾收集器和复制数据，而不需要使用 IntPtr 和不安全的代码，ref 关键字的增强功能提供了极大的帮助。

ReferenceSemantics 示例使用了如下名称空间：

System

System.Linq

看看下面的 Data 类。该类包含变量名称为_anumber 的值类型 int，它在构造函数中初始化。方法 Show 将数字的当前值写入控制台。最有趣的部分是 GetNumber 方法。在实现代码中，变量_anumber 使用 ref 关键字返回，以返回对它的引用。这是由 GetNumber 的返回类型的声明实现的；它是 ref int 类型的声明，返回一个对 int 的引用。GetReadonlyNumber 方法是一个 ref readonly int 返回的方法。ref readonly 是 C# 7.2 中新增的，通过引用返回值类型，但是不允许由调用者改变(代码文件 ReferenceSemantics/Data.cs)：

```
public class Data
{
  public Data(int anumber) => _anumber = anumber;
  private int _anumber;

  public ref int GetNumber() => ref _anumber;

  public ref readonly int GetReadonlyNumber() => ref _anumber;

  public void Show() => Console.WriteLine($"Data: {_anumber}");
}
```

下面使用 Data 类并调用 GetNumber 方法。该方法声明为返回 ref int，但是在下面的代码片段中，结果写到一个 int 中。n 是一个本地变量，它保存了一个 int，GetNumber 的结果会复制到这个变量中。更改本地变量的值时，Data 类内的数据不会更改(代码文件 ReferenceSemantics/Program.cs)：

```
static void UseMember()
{
  Console.WriteLine(nameof(UseMember));
  var d = new Data(11);
  int n = d.GetNumber();
  n = 42;
  d.Show();
  Console.WriteLine();
}
```

运行应用程序时，输出显示，Data 类在本地变量更改之后仍然包含初始化的数据：

```
UseMember
Data: 11
```

在方法 UseRefMember 的实现中做一个小小的更改，调用 GetNumber 方法，返回一个 ref，它在方法之前指定 ref 关键字，变量 n 指定为 ref local，因此它在 Data 类中直接引用_anumber。也可以用 ref readonly 修饰符声明本地变量。方法 GetNumber 返回 ref int 的结果可以分配给 ref readonly int，这保证不能更改变量 n2。编译器会抱怨 n2 是否会被更改(代码文件 ReferenceSemantics/Program.cs)：

```
static void UseRefMember()
{
  Console.WriteLine(nameof(UseRefMember));
  var d = new Data(11);
  ref int n = ref d.GetNumber();
  n = 42;
  d.Show();

  ref readonly int n2 = d.GetNumber();
  // n2 = 42; // not allowed - it's readonly!
  Console.WriteLine();
}
```

使用此更改运行应用程序时，Data 类中的数据将更改。不需要使用 ntPtr 和不安全的代码，也可以快速直接地访问：

```
UseRefMember
Data: 42
```

接下来，调用方法 GetReadonlyNumber。这个方法返回 ref readonly int。可以将结果赋给一个 int。把 ref 赋予 int 会建立一个副本，副本可以更改，但不会更改原始副本。将结果分配给 ref readonly int，会通过引用传递结果，但结果不能更改(代码文件 ReferenceSemantics/Program.cs)：

```
static void UseReadonlyRefMember()
{
  Console.WriteLine(nameof(UseReadonlyRefMember));
  var d = new Data(11);
  int n = d.GetReadonlyNumber();  // create a copy
  n = 42;
  d.Show();

  // ref int n = d.GetReadonlyNumber(); // not allowed
  ref readonly int n2 = ref d.GetReadonlyNumber();
  // n2 = 42; // not allowed
  Console.WriteLine();
}
```

该方法的结果是一个不变的 Data 成员：

```
UseRefMember
Data: 11
```

### 17.6.1　传递 ref 和返回 ref

下面是另一个例子：传递一个 ref int 并返回一个 ref int。 Max 方法通过 ref 接收 x 和 y 参数，并通过 ref 返回这两个值的较高者(代码文件 ReferenceSemantics/Program.cs)：

```
static ref int Max(ref int x, ref int y)
{
  if (x > y) return ref x;
  else return ref y;
}
```

如果不需要复制变量 x 和 y，将它们传递给方法 Max，则可以快速返回较高的值。如果经常调用此方法，这将非常有用：

```
static void UseMax()
{
  Console.WriteLine(nameof(UseMax));
  int x = 4, y = 5;
  ref int z = ref Max(ref x, ref y);
  Console.WriteLine($"{z} is the max of {x} and {y}");
  //...
}
```

返回的消息如下：

```
5 is the max of 4 and 5
```

返回引用是很快速的，因为幕后只使用指针。但是，这也意味着可以更改引用指向的原始项。例如，改变引用 x 或 y 中数据的变量 z，根据较大的值，也改变了原始变量的值：

```
static void UseMax()
{
  //...
  z = x + y;
  Console.WriteLine($"y after changing z: {y}");

  Console.WriteLine();
}
```

运行这个程序时，可以看到 y 现在有了被分配的值。

```
y after changing z: 9
```

### 17.6.2　ref 和数组

另一个展示 ref return 和 ref local 特性的示例使用了该关键字与数组。类 Container 定义了 int[]类型的成员(它们在构造函数中初始化)。GetItem 方法通过引用返回数组的一个项。这允许在容器数组中直接使用快速路径(代

码文件 ReferenceSemantics/Container.cs)：

```
public class Container
{
  public Container(int[] data) => _data = data;
  private int[] _data;

  //...

  public ref int GetItem(int index) => ref _data[index];

  public void ShowAll()
  {
    Console.WriteLine(string.Join(", ", _data));
    Console.WriteLine();
  }
}
```

当使用这个 Container 时，一个包含 10 项列表的样本数组被传递给构造函数。第 4 项从 GetItem 方法中检索，此项更改为 33，最后所有项都使用 ShowAll 方法写入控制台(代码文件 ReferenceSemantics/Program.cs)：

```
private static void UseItemOfContainer()
{
  Console.WriteLine(nameof(UseItemOfContainer));
  var c = new Container(Enumerable.Range(0, 10).Select(x => x).ToArray());
  ref int item = ref c.GetItem(3);
  item = 33;
  c.ShowAll();
  Console.WriteLine();
}
```

运行应用程序时，可以看到第 4 项从外部更改：

```
UseItemOfContainer
0, 1, 2, 33, 4, 5, 6, 7, 8, 9
```

下面看看添加 GetData 方法除了处理数组项之外，还可以处理完整的数组。该方法返回一个对数组本身的引用(代码文件 ReferenceSemantics/Container.cs)：

```
public class Container
{
  //...

  public ref int[] GetData() => ref _data;

  //...
}
```

使用 Container 类的 GetData 方法，将返回数组的引用，并将其写入 ref 本地变量 d1 中。一个包含三个元素的新数组被分配给这个变量(代码文件 ReferenceSemantics/Program.cs)：

```
private static void UseArrayOfContainer()
{
  Console.WriteLine(nameof(UseArrayOfContainer));
  var c = new Container(Enumerable.Range(0, 10).Select(x => x).ToArray());
  ref int[] d1 = ref c.GetData();
  d1 = new int[] { 4, 5, 6 };
  c.ShowAll();
  Console.WriteLine();
}
```

因为返回对数组的引用，所以可以替换完整的数组。容器现在包含新创建的数组，其中包含元素 4、5 和 6：

```
UseArrayOfContainer
4, 5, 6
```

**注意：**

用于 ref returns 和 ref locals 的 ref 关键字需要在返回引用时保持活跃。例如，只要在引用类型中包含值类型，就可以返回对值类型的引用，这样它们就在托管的堆中。使用结构，不能定义方法来返回结构成员的引用。可以将对结构的引用返回为引用，如 Max 方法所示。这些值类型在方法返回时应保证是活跃的，因为它们是由等待返回方法的调用者传递的。

**注意：**

第 3 章介绍了使用 ref、out 和 in 修饰符定义参数。这些修饰符在引用语义方面也很重要。使用 C# 7.2 的新参数 in 和值类型，指定值类型是通过引用传递的(类似于通过参数使用 ref 关键字)，但是不允许更改它。对于参数，in 类似于 ref readonly。

## 17.7　Span<T>

第 3 章介绍了创建引用类型(类)和值类型(结构)。类的实例存储在托管堆上。结构的值可以存储在堆栈上，或者当使用装箱时，可以存储在托管堆上。现在我们有了另一种类型：一种只能在堆栈上存储其值的类型，而不会在堆上存储，有时称为类 ref 类型。这种类型的装箱是不可能的。这样的类型用 ref truct 关键字声明。使用 ref struct 提供了一些额外的行为和限制。限制如下：

- 它们不能添加为数组项。
- 它们不能用作泛型类型参数。
- 它们不能装箱。
- 它们不能是静态字段。
- 它们只能是类 ref 类型的实例字段。

在本节中，Span<T>和 ReadOnlySpan<T>是类似于 ref 的类型。这些类型已经在讨论数组扩展方法的第 7 章中介绍了，在第 9 章中介绍了字符串的扩展方法。这里介绍的附加特性包括在托管堆、堆栈和本机堆上引用数据。

### 17.7.1　Span 引用托管堆

Span 可以引用托管堆上的内存，如第 7 和第 9 章所示。在下面的代码片段中，创建了一个数组，并使用扩展方法 AsSpan 创建了一个新的 Span，它引用托管堆上数组的内存。创建在变量 span1 中引用的 Span 之后，创建 Span 的一个切片，其中用值 42 填充。下一个 Console.WriteLine 将 span1 的值写入控制台(代码文件 SpanSample/Program.cs)：

```
private static void SpanOnTheHeap()
{
  Console.WriteLine(nameof(SpanOnTheHeap));
  Span<int> span1 = (new int[] { 1, 5, 11, 71, 22, 19, 21, 33 }).AsSpan();

  span1.Slice(start: 4, length: 3).Fill(42);

  Console.WriteLine(string.Join(", ", span1.ToArray()));

  Console.WriteLine();
}
```

运行应用程序时，可以看到在 span 的切片中，用 42 填充的 span1 的输出：

```
SpanOnTheHeap
1, 5, 11, 71, 42, 42, 42, 33
```

### 17.7.2　Span 引用栈

Span 可以用来引用堆栈上的内存。在堆栈上引用单个变量并不像引用一个内存块那样有趣；这就是为什么下面的代码片段使用 stackalloc 关键字的原因。stackalloc 返回一个 long*，它要求将方法 SpanOnTheStack 声明为 unsafe，而 Span 类型的构造函数允许传递一个指针和表示该指针大小的附加参数。接下来，变量 span1 与索引器一起使用，以填充每个条目(代码文件 SpanSample/Program.cs)：

```
private static unsafe void SpanOnTheStack()
{
  Console.WriteLine(nameof(SpanOnTheStack));

  long* lp = stackalloc long[20];
```

```
  var span1 = new Span<long>(lp, 20);

  for (int i = 0; i < 20; i++)
  {
    span1[i] = i;
  }

  Console.WriteLine(string.Join(", ", span1.ToArray()));
  Console.WriteLine();
}
```

运行该程序时，以下输出显示了在堆栈上初始化数据的 Span：

```
SpanOnTheStack
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19
```

### 17.7.3 Span 引用本机堆

Span 的一个重要特征是它们也可以引用本机堆上的内存。本机堆上的内存通常是从本机 API 上分配的。在下面的代码片段中，Marshal 类的 AllocHGlobal 方法用于在本机堆上分配 100 个字节。Marshal 类返回一个 IntPtr 类型的指针。为了直接访问 int*，调用了 IntPtr 的 ToPointer 方法。这是 Span 类的构造函数所需要的指针。在此内存中写入 int 值，应注意需要多少字节。当 int 包含 32 位时，字节数除以 4，并移动两位。在此之后，通过调用 Span 的 Fill 方法来填充本机内存。在 for 循环中，从 Span 中引用的每个项都写到控制台(代码文件 SpanSample/Program.cs)：

```
private static unsafe void SpanOnNativeMemory()
{
  Console.WriteLine(nameof(SpanOnNativeMemory));
  const int nbytes = 100;
  IntPtr p = Marshal.AllocHGlobal(nbytes);
  try
  {
    int* p2 = (int*)p.ToPointer();
    Span<int> span = new Span<int>(p2, nbytes >> 2);
    span.Fill(42);

    int max = nbytes >> 2;
    for (int i = 0; i < max; i++)
    {
      Console.Write($"{span[i]} ");
    }
    Console.WriteLine();
  }
  finally
  {
    Marshal.FreeHGlobal(p);
  }
  Console.WriteLine();
}
```

运行应用程序时，存储在本机堆中的值将写入控制台：

```
SpanOnNativeMemory
42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42
```

**注意：**

*使用 Span 访问本机内存和堆栈，需要使用不安全的代码，因为通过传递一个指针来创建 Span 和分配内存。在初始化之后，使用 Span，就不再需要不安全的代码。*

### 17.7.4 Span 扩展方法

对于 Span 类型，定义了扩展方法，使其更容易使用这种类型。下面的代码片段演示了 Overlaps、Reverse 和 IndexOf 方法的使用。使用 Overlaps 方法，检查用于调用此扩展方法的 span 是否与该参数传递的 span 重叠。Reverse 方法反转了 span 的内容。IndexOf 方法返回用参数传递的 span 的索引(代码文件 SpanSample/Program.cs)：

```
private static void SpanExtensions()
{
```

```
    Console.WriteLine(nameof(SpanExtensions));
    Span<int> span1 = (new int[] { 1, 5, 11, 71, 22, 19, 21, 33 }).AsSpan();
    Span<int> span2 = span1.Slice(3, 4);
    bool overlaps = span1.Overlaps(span2);
    Console.WriteLine($"span1 overlaps span2: {overlaps}");
    span1.Reverse();
    Console.WriteLine($"span1 reversed: {string.Join(", ", span1.ToArray())}");
    Console.WriteLine($"span2 (a slice) after reversing span1: " +
      $"{string.Join(", ", span2.ToArray())}");
    int index = span1.IndexOf(span2);
    Console.WriteLine($"index of span2 in span1: {index}");
    Console.WriteLine();
}
```

运行这个程序，产生如下输出：

```
SpanExtensions
span1 overlaps span2: True
span1 reversed: 33, 21, 19, 22, 71, 11, 5, 1
span2 (a slice) after reversing span1: 22, 71, 11, 5
index of span2 in span1: 3
```

为 Span 类型定义的其他扩展方法是 StartWith，用于检查一个 Span 是否以另一个 Span 的序列开始，SequenceEqual 用于比较两个 Span 的序列，SequenceCompareTo 用于对序列排序，LastIndexOf 返回从 Span 末尾处开始的第一个匹配索引。

## 17.8　平台调用

并不是 Windows API 调用的所有特性都可用于.NET。旧的 Windows API 调用是这样，新功能也是这样。也许开发人员会编写一些 DLL，导出非托管的方法，在 C#中使用它们。

要重用一个非托管库，其中不包含 COM 对象，只包含导出的功能，就可以使用平台调用(P/Invoke)。有了 P/Invoke，CLR 会加载 DLL，其中包含应调用的函数，并编组参数。

要使用非托管函数，首先必须确定导出的函数名。为此，可以使用 dumpbin 工具和 /exports 选项。例如，命令：

```
dumpbin /exports c:\windows\system32\kernel32.dll | more
```

列出 DLL kernel32.dll 中所有导出的函数。这个示例使用 Windows API 函数 CreateHardLink 来创建到现有文件的硬链接。使用此 API 调用，可以用几个文件名引用相同的文件，只要文件名在一个硬盘上即可。这个 API 调用不能用于.NET Core，因此必须使用平台调用。

为了调用本机函数，必须定义一个参数数量相同的 C#外部方法，用非托管方法定义的参数类型必须用托管代码映射类型。

在 C++中，Windows API 调用 CreateHardLink 有如下定义：

```
BOOL CreateHardLink(
LPCTSTR lpFileName,
LPCTSTR lpExistingFileName,
LPSECURITY_ATTRIBUTES lpSecurityAttributes);
```

这个定义必须映射到.NET 数据类型上。非托管代码的返回类型是 BOOL；它仅映射到 bool 数据类型。LPCTSTR 定义了一个指向 const 字符串的 long 指针。Windows API 给数据类型使用 Hungarian 命名约定。LP 是一个 long 指针，C 是一个常量，STR 是以 null 结尾的字符串。T 把类型标志为泛型类型，根据编译器设置为 32 位还是 64 位，该类型解析为 LPCSTR(ANSI 字符串)或 LPWSTR(宽 Unicode 字符串)。C 字符串映射到.NET 类型 String。LPSECURITY_ATTRIBUTES 是一个 long 指针，指向 SECURITY_ATTRIBUTES 类型的结构。因为可以把 NULL 传递给这个参数，所以把这种类型映射到 IntPtr 是可行的。该方法的 C#声明必须用 extern 修饰符标记，因为在 C#代码中，这个方法没有实现代码。相反，该方法的实现代码在 DLL kernel32.dll 中，它用属性[DllImport]引用。.NET 声明 CreateHardLink 的返回类型是 bool，本机方法 CreateHardLink 返回一个布尔值，所以需要一些额外的澄清。因为 C++有不同的 Boolean 数据类型(例如，本机 bool 和 Windows 定义的 BOOL 有不同的值)，所以特性[MarshalAs]指定.NET 类型 bool 应该映射为哪个本机类型：

```
[DllImport("kernel32.dll", SetLastError="true",
  EntryPoint="CreateHardLink", CharSet=CharSet.Unicode)]
[return: MarshalAs(UnmanagedType.Bool)]
public static extern bool CreateHardLink(string newFileName,
  string existingFilename, IntPtr securityAttributes);
```

**注意：**
网站 http://www.pinvoke.net 非常有助于从本机代码到托管代码的转换。

可以用[DllImport]属性指定的设置在表 17-2 中列出。

表 17-2

| DllImport 属性或字段 | 说　明 |
|---|---|
| EntryPoint | 可以给函数的 C#声明指定与非托管库不同的名称。非托管库中方法的名称在 EntryPoint 字段中定义 |
| CallingConvention | 根据编译器或用来编译非托管函数的编译器设置，可以使用不同的调用约定。调用约定定义了如何处理参数，把它们放在堆栈的什么地方。可以设置一个可枚举的值，来定义调用约定。Windows API 在 Windows 操作系统上通常使用 StdCall 调用约定，在 Windows CE 上使用 Cdecl 调用约定。把值设置为 CallingConvention.Winapi，可让 Windows API 用于 Windows 和 Windows CE 环境 |
| CharSet | 字符串参数可以是 ANSI 或 Unicode。通过 CharSet 设置，可以定义字符串的管理方式。用 CharSet 枚举定义的值有 Ansi、Unicode 和 Auto.CharSet。Auto 在 Windows NT 平台上使用 Unicode，在微软的旧操作系统上使用 ANSI |
| SetLastError | 如果非托管函数使用 Windows API SetLastError 设置一个错误，就可以把 SetLastError 字段设置为 true。这样，就可以使用 Marshal. GetLastWin32Error 读取后面的错误号 |

为了使 CreateHardLink 方法更易于在.NET 环境中使用，应该遵循如下规则：

- 创建一个内部类 NativeMethods，来包装平台调用的方法调用。
- 创建一个公共类，给.NET 应用程序提供本机方法的功能。
- 使用安全特性来标记所需的安全。

在接下来的例子中，类 FileUtility 中的公共方法 CreateHardLink 可以由.NET 应用程序使用。这个方法的文件名参数，与本机 Windows API 方法 CreateHardLink 的顺序相反。第一个参数是现有文件的名称，第二个参数是新的文件。这类似于框架中的其他类，如 File.Copy。

因为第三个参数用来传递新文件名的安全特性，此实现代码不使用它，所以公共方法只有两个参数。返回类型也改变了。它不通过返回 false 值来返回一个错误，而是抛出一个异常。如果出错，非托管方法 CreateHardLink 就用非托管 API SetLastError 设置错误号。要从.NET 中读取这个值，[DllImport] 字段 SetLastError 设置为 true。在托管方法 CreateHardLink 中，错误号是通过调用 Marshal.GetLastWin32Error 读取的。要从这个号中创建一个错误消息，应使用 System.ComponentModel 名称空间中的 Win32Exception 类。这个类通过构造函数接受错误号，并返回一个本地化的错误消息。如果出错，就抛出 IOException 类型的异常，它有一个类型 Win32Exception 的内部异常。应用公共方法 CreateHardLink 的 FileIOPermission 特性，检查调用程序是否拥有必要的许可(代码文件 PInvokeSampleLib/NativeMethods.cs)：

```
[SecurityCritical]
internal static class NativeMethods
{
  [DllImport("kernel32.dll", SetLastError = true,
    EntryPoint = "CreateHardLinkW", CharSet = CharSet.Unicode)]
  [return: MarshalAs(UnmanagedType.Bool)]
    private static extern bool CreateHardLink(
    [In, MarshalAs(UnmanagedType.LPWStr)] string newFileName,
    [In, MarshalAs(UnmanagedType.LPWStr)] string existingFileName,
    IntPtr securityAttributes);
  internal static void CreateHardLink(string oldFileName,
    string newFileName)
  {
```

```
        if (!CreateHardLink(newFileName, oldFileName, IntPtr.Zero))
        {
          var ex = new Win32Exception(Marshal.GetLastWin32Error());
          throw new IOException(ex.Message, ex);
        }
      }
    }

    public static class FileUtility
    {
      [FileIOPermission(SecurityAction.LinkDemand, Unrestricted = true)]
      public static void CreateHardLink(string oldFileName,
        string newFileName)
      {
        NativeMethods.CreateHardLink(oldFileName, newFileName);
      }
    }
```

这个库使用如下依赖项和名称空间：

**依赖项**

System.Security.Permissions

**名称空间**

System

System.IO

System.Runtime.InteropServices

System.Security

System.Security.Permissions

> **警告：**
> PlatformInvoke 示例在 Linux 上成功编译，但没有运行，因为在 Linux 操作系统上找不到库 kernel32.dll。

现在可以使用这个类轻松地创建硬链接。如果程序的第一个参数传递的文件不存在，就会得到一个异常，提示“系统无法找到指定的文件”。如果文件存在，就得到一个引用原始文件的新文件名。很容易验证它：在一个文件中改变文本，它就会出现在另一个文件中(代码文件 PInvokeSample/Program.cs)：

```
class Program
{
  static void Main(string[] args)
  {
    if (args.Length != 2)
    {
      Console.WriteLine("usage: PInvokeSample " +
        "existingfilename newfilename");
      return;
    }
    try
    {
      FileUtility.CreateHardLink(args[0], args[1]);
    }
    catch (IOException ex)
    {
      Console.WriteLine(ex.Message);
    }
  }
}
```

在Windows上调用本地方法时，通常必须使用Windows句柄。Windows句柄是一个32位或64位值，根据句柄类型，不允许使用一些值。在.NET 1.0中，句柄通常使用IntPtr结构，因为可以用这种结构设置每一个可能的32位值。然而，对于一些句柄类型，这会导致安全问题，可能还会出现线程竞态条件，在终结阶段泄漏句柄。所以.NET 2.0引入了SafeHandle类。SafeHandle类是一个抽象的基类，用于每个Windows句柄。Microsoft.Win32.SafeHandles名称空间中的派生类是SafeHandleZeroOrMinus OneIsInvalid和SafeHandleMinusOneIsInvalid。顾名思义，这些类不接受无效的0或-1值。进一步派生的句柄类型是SafeFileHandle、SafeWaitHandle、SafeNCryptHandle

和SafePipeHandle，可以供特定的Windows API调用使用。

例如，为了映射 Windows API CreateFile，可以使用以下声明返回一个 SafeFileHandle。当然，通常可以使用.NET 类 File 和 FileInfo。

```
[DllImport("Kernel32.dll", SetLastError = true,
  CharSet = CharSet.Unicode)]
internal static extern SafeFileHandle CreateFile(
  string fileName,
  [MarshalAs(UnmanagedType.U4)] FileAccess fileAccess,
  [MarshalAs(UnmanagedType.U4)] FileShare fileShare,
  IntPtr securityAttributes,
  [MarshalAs(UnmanagedType.U4)] FileMode creationDisposition,
  int flags,
  SafeFileHandle template);
```

## 17.9 小结

要想成为真正优秀的 C#程序员，必须牢固掌握存储单元和垃圾收集的工作原理。本章描述了 CLR 管理以及在堆和栈上分配内存的方式，讨论了如何编写正确地释放非托管资源的类，并介绍如何在 C#中使用指针，这些都是很难理解的高级主题，初学者常常不能正确实现。至少本章有助于理解如何使用 IDisposable 接口和 using 语句释放资源。

本章还介绍了 C# 7.0 和 C#7.2 通过引用传递值和通过引用返回值的增强功能，特别是 ref return 和 ref locals,，以及使用 ref readonly 修饰符。

下一章将介绍 Visual Studio 2017 的所有功能。

# 第18章

# Visual Studio 2017

**本章要点**

- 使用 Visual Studio 2017
- 创建和使用项目
- 调试
- 用 Visual Studio 进行重构
- 使用不同技术工作(UWP、ASP.NET Core 等)
- 分析应用程序
- 通过 Docker 创建和使用容器

**本章源代码下载地址(wrox.com)：**

打开 www.wrox.com 的 Download Code 选项卡可下载本章源代码。源代码也可以在 VisualStudio 目录的 https://github.com/ProfessionalCSharp/ProfessionalCSharp7 中找到。本章代码分为以下几个主要的示例文件：

- DockerSample
- WebAppWithVS

## 18.1 使用 Visual Studio 2017

到目前为止，你应该已经对 C#语言比较熟悉，并准备开始学习本书的应用部分。在这些章节中会介绍如何使用 C#编写各种应用程序。但在学习之前，需要理解如何使用 Visual Studio 和.NET 环境提供的一些功能，使程序达到最佳效果。

本章讲解在实际工作中，如何在.NET 环境中编程。介绍主要的开发环境 Visual Studio，该环境用于编写、编译、调试和优化 C#程序，并且为编写优秀的应用程序提供指导。Visual Studio 是主要的 IDE，用于多种目的，包括编写 ASP.NET 和 ASP.NET Core Web 应用程序、Windows Presentation Foundation (WPF) 应用程序、用于 Universal Windows Platform (UWP)的应用程序、由 ASP.NET Web 创建的访问服务。

本章还探讨如何构建目标框架为.NET Core 的应用程序。

Visual Studio 2017 是一个全面集成的开发环境。编写、调试、编译代码以生成一个程序集的整个过程被设计得尽可能容易。这意味着 Visual Studio 是一个非常全面的多文档界面应用程序，在该环境中可以完成所有代

码开发的相关事情。它具有以下特性：

- **文本编辑器**——使用这个编辑器，可以编写 C#(还有 Visual Basic、C++、F#、JavaScript、XAML、JSON 以及 SQL)代码。这个文本编辑器是非常先进的。例如，当用户输入时，它会用缩进代码行自动布局代码，匹配代码块的开始和结束括号，以及使用颜色编码关键字。它还会在用户输入时检查语法，并用下划线标识导致编译错误的代码，这也称为设计时的调试。另外，它具有 IntelliSense 功能，当开始输入时它会自动显示类、字段或方法的名称。开始输入方法参数时，它也会显示可用重载的参数列表。图 18-1 用 UWP 应用程序展示了 IntelliSense 功能。这个对话框在 Visual Studio 2017 中有一个新特性：可以使用底部的按钮选择只查看属性、事件或方法。这对大型团队列表有很大帮助。

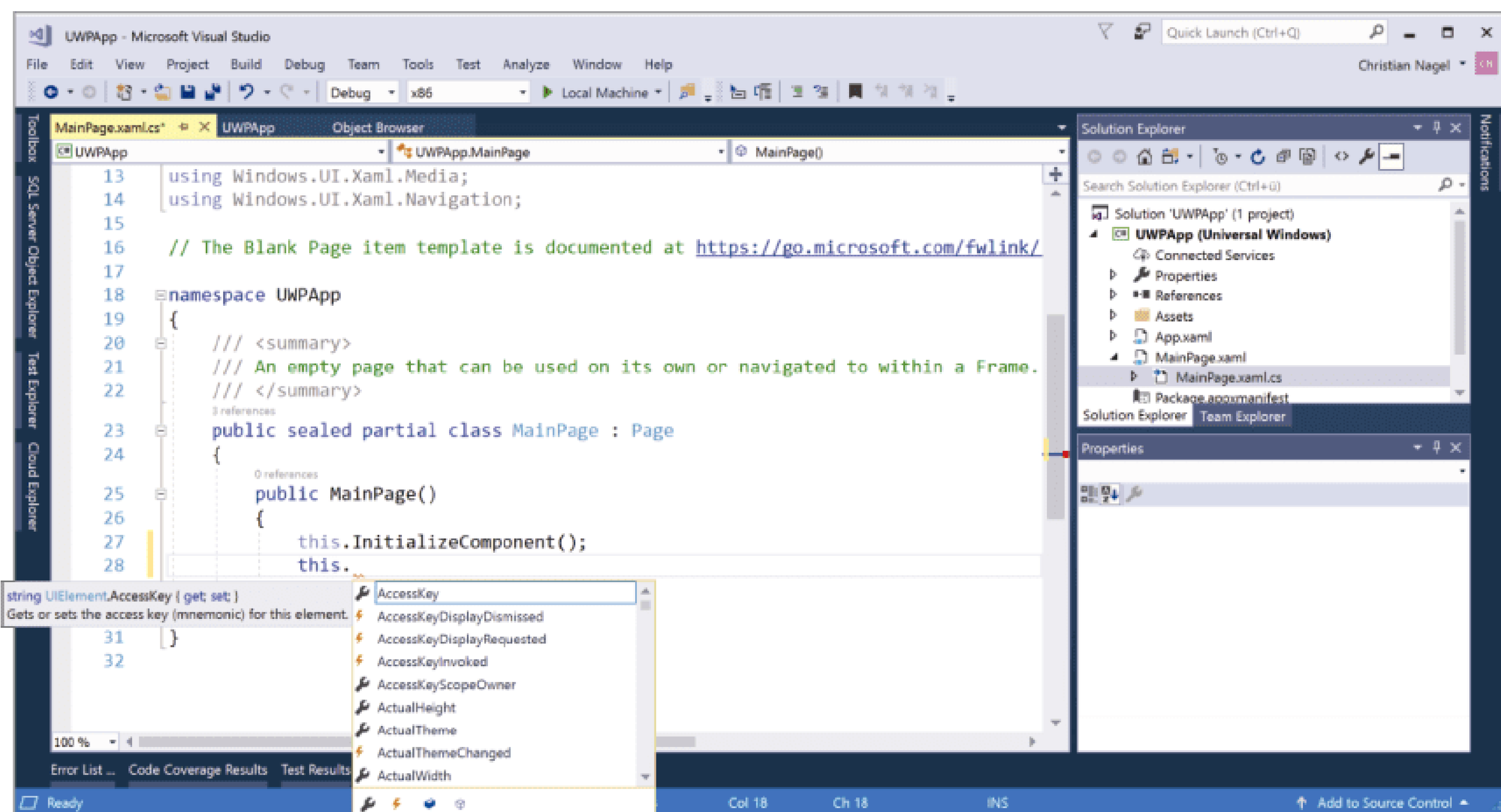


图 18-1

**注意：**

如果需要 IntelliSense 的列表框，或者因为其他原因该列表框不见了，可以按下 Ctrl+Space 组合键找回该列表框。如果希望看到 IntelliSense 框下面的代码，可以按住 Ctrl 按钮。

- **设计视图编辑器**——这个编辑器允许在项目中放置用户界面控件和数据绑定控件；Visual Studio 会在项目中自动将必需的 C#代码添加到源文件中，来实例化这些控件(这是可能的，因为所有.NET 控件都是具体基类的实例)。
- **支持窗口**——这些窗口允许查看和修改项目的各个方面，例如源代码中的类、Windows Forms 和 Web Forms 类的可用属性(以及它们的启动值)。也可以使用这些窗口来指定编译选项，例如代码需要引用的程序集。
- **集成的调试器**——从编程的本质讲，第一次试运行时，代码可能会无法正常运行。可能第二次或者第三次都无法正常运行。Visual Studio 无缝地链接到一个调试器中，允许设置断点，监视集成环境中的变量。
- **集成的 MSDN 帮助**——Visual Studio 允许在 IDE 中访问 MSDN 文档。例如，如果使用文本编辑器时不太确定一个关键字的含义，只需要选择该关键字并按 F1 键，Visual Studio 将会访问 https://docs.microsoft.com 并展示相关主题。同样，如果不确定某个编译错误是什么意思，可以选择错误消息并按 F1 键，调出 MSDN 文档，查看该错误的演示。
- **访问其他程序**——Visual Studio 也可以访问一些其他实用程序，在不退出集成开发环境的情况下，就可以检查和修改计算机或网络的相关方面。可以用这些实用工具检查运行的服务和数据库连接，直接查看 SQL Server 表，浏览 Microsoft Azure Cloud 服务，甚至用一个 Web 浏览器窗口来浏览 Web。

- **Visual Studio 扩展**　Visual Studio 的一些扩展已经在 Visual Studio 的正常安装过程中安装好了，Microsoft 和第三方还提供了更多的扩展。这些扩展允许分析代码，提供项目或项模板，访问其他服务等。使用.NET 编译器平台，与 Visual Studio 工具的集成会更简单。

Visual Studio 的最新版本有一些有趣的改进。一个主要部分是用户界面，另一个主要部分是后台功能和.NET 编译器平台。

对于用户界面，Visual Studio 2010 基于 WPF 重新设计了外壳，而不是基于原生的 Windows 控件。Visual Studio 2012 的界面在此基础上又有了一些变化，尤其是用户界面更关注主要工作区——编辑器，允许直接在代码编辑器中完成更多的工作，而不需要使用许多其他工具。当然，还需要代码编辑器之外的一些工具，但更多的功能内置于几个工具中，所以减少了通常需要的工具数量。在 Visual Studio 2017 中，改进了一些 UI 功能。可以立即在 Visual Studio 安装程序中看到 first UI 增强，它从 Windows 8 磁贴的设计中获得了一些灵感，更容易地选择工作负载(参见图 18-2)。

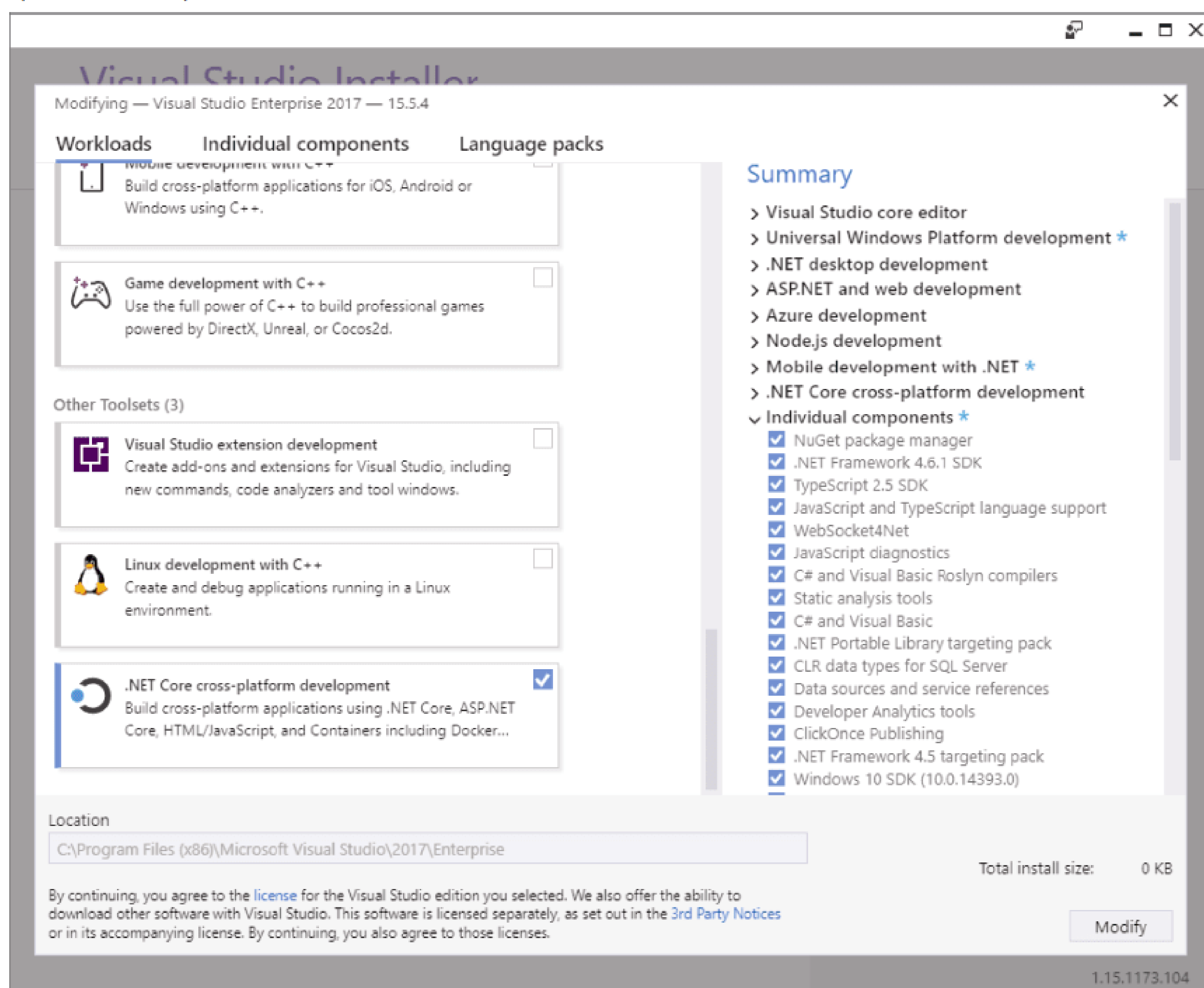



图 18-2

有了.NET 编译器平台(代码名称是 Roslyn)，.NET 编译器完全重写了，它现在集成了编译器管道的功能，例如语法分析、语义分析、绑定和代码输出。Microsoft 基于此重写了许多 Visual Studio 集成工具。代码编辑器、智能感知和重构都基于.NET 编译器平台。

对于 XAML 代码编辑，Visual Studio 2010 和 Blend for Visual Studio 共享相同的编辑器引擎。不仅代码引擎是一样的：Visual Studio 2013 从 Blend 中得到了 XAML 引擎，自从 Blend for Visual Studio 2015 以来，Blend 获得了 Visual Studio 的外壳。启动 Blend for Visual Studio 2017，会看到它类似于 Visual Studio，可以立即开始使用它。

Visual Studio 的另一项改进是搜索。Visual Studio 有许多命令和功能，常常很难找到需要的菜单或工具栏按钮。只要在 Quick Launch 中输入所需命令的一部分，就可以看到可用的选项。Quick Launch 位于窗口的右上角(见图 18-3)。搜索功能还可以从其他地方找到，如工具栏、解决方案资源管理器、代码编辑器(可以按 Ctrl+F 组合键来调用)以及引用管理器上的程序集等。

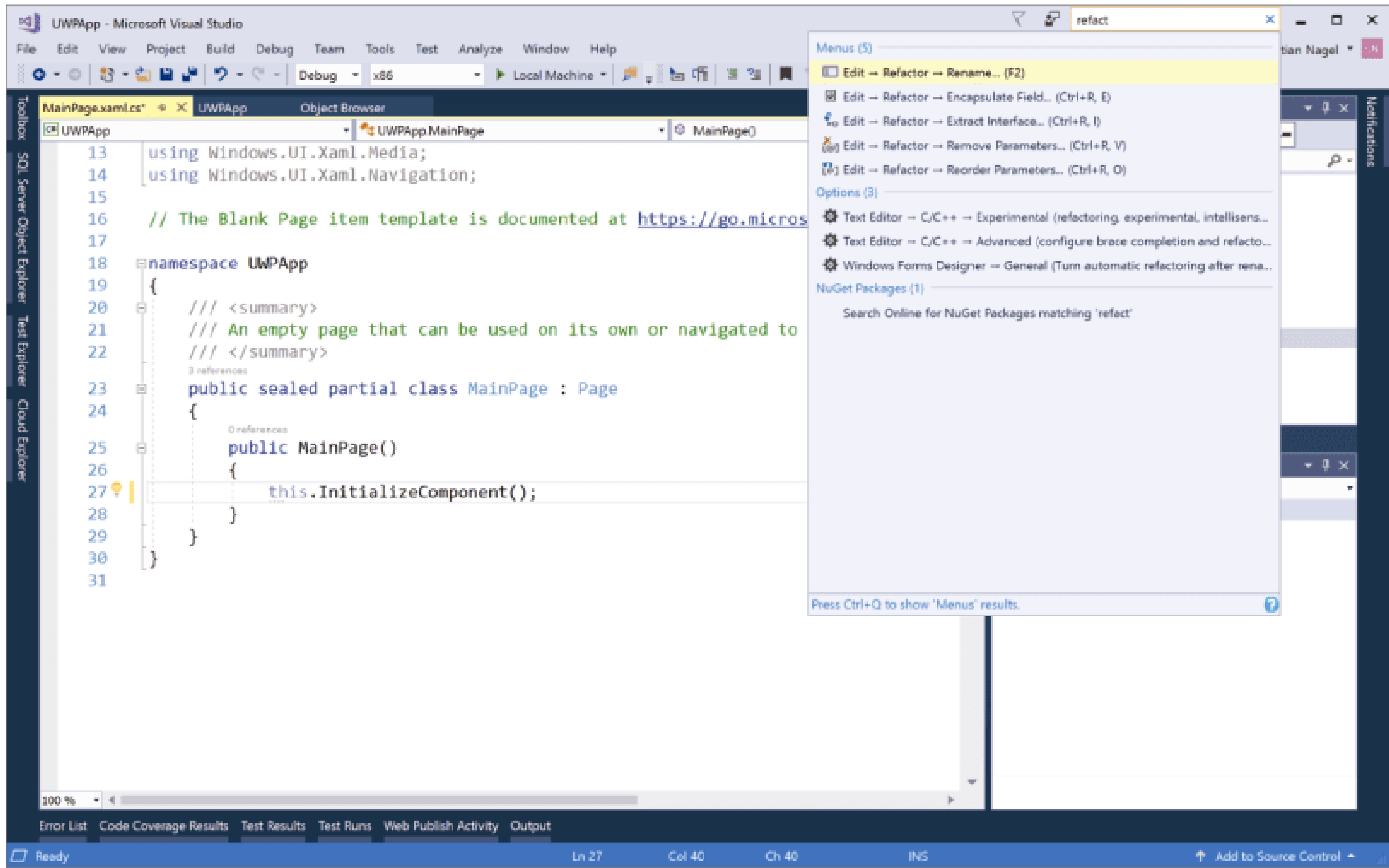



图 18-3

## 18.1.1　Visual Studio 的版本

Visual Studio 2017 提供了多个版本。最便宜的是 Visual Studio 2017 Community 版，这个版本在某些情况下是免费的！它对个人开发者、开源项目、学术研究、教育和小型专业团队是免费的。

可供购买的是 Professional 和 Enterprise 版。只有 Enterprise 版包含所有功能。Enterprise 版独享的功能有 IntelliTrace(智能跟踪)、负载测试和一些架构工具。微软的 Fakes 框架(隔离单元测试)只能用于 Visual Studio Enterprise 版。本章介绍 Visual Studio 2017 包含的一些功能，这些功能仅适用于特定版本。有关 Visual Studio 2017 各个版本中功能的详细信息，请参考 http://www.visualstudio.com/vs/compare/。

## 18.1.2　Visual Studio 设置

当第一次运行 Visual Studio 时，需要选择一个符合环境的设置集，例如 General Development、Visual Basic、Visual C#、Visual C++或 Web Development。这些不同的设置反映了用于这些语言的不同工具。在微软平台上编写应用程序时，可以使用不同的工具创建 Visual Basic、C++和 Web 应用程序。同样，Visual Basic、Visual C++和 Visual InterDev 具有完全不同的编程环境、设置和工具选项。现在，可以使用 Visual Studio 为所有这些技术创建应用程序，但 Visual Studio 仍然提供了快捷键，可以根据 Visual Basic、Visual C++和 Visual InterDev 选择。当然，也可以选择特定的 C#设置。

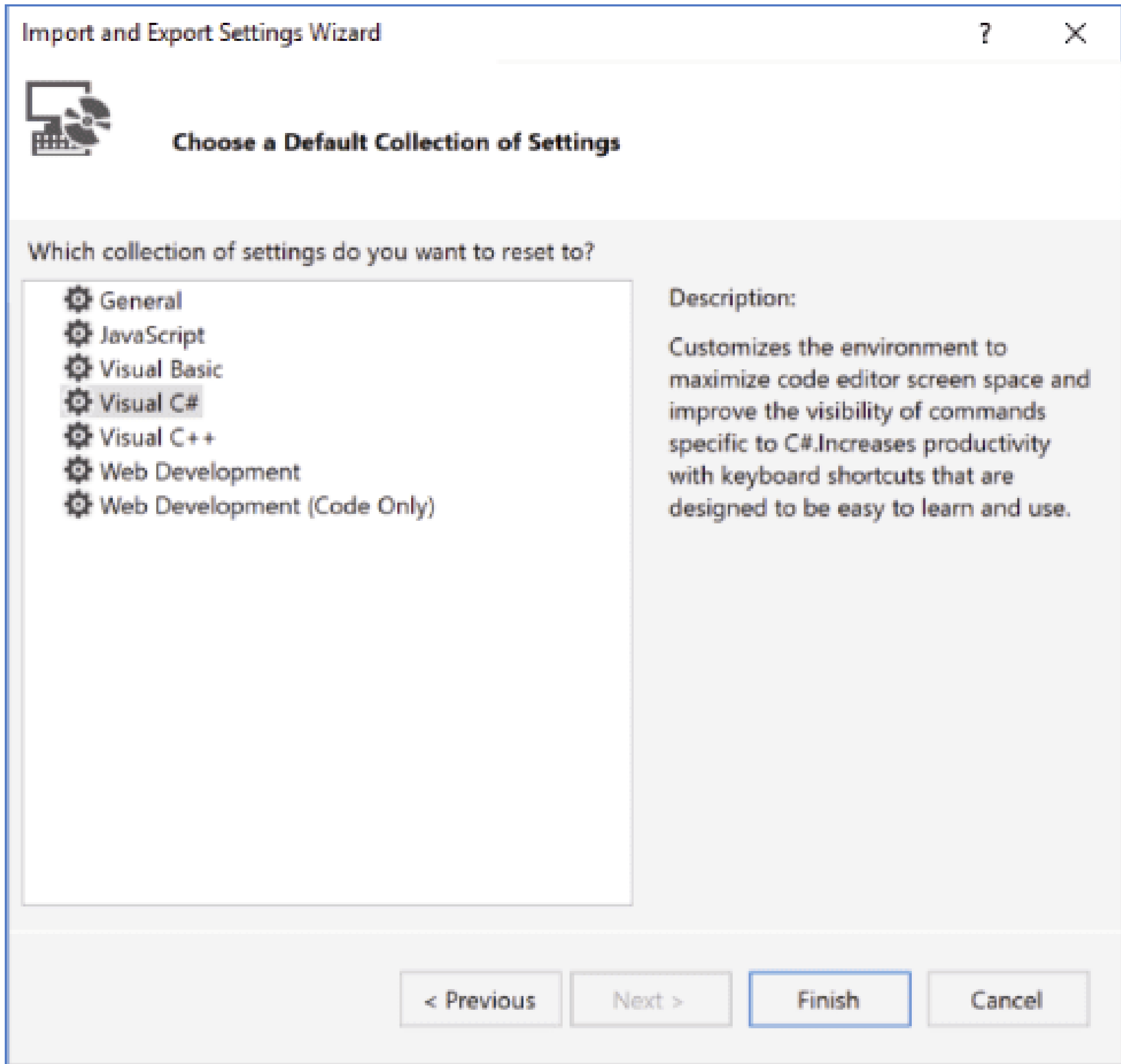



图 18-4

在选择了设置的主类别，确定了键盘快捷键、菜单和工具窗口的位置后，就可以通过 Tools | Customize…(工具栏和命令)和 Tools | Options…(在此可以找到所有工具的设置)，来改变每个设置。也可以重置设置集，方法是使用 Tools | Import and Export Settings 调用一个向导，来选择一个新的默认设置集(如图 18-4 所示)。

接下来的小节贯穿一个项目的创建、编码和调试过程，展示 Visual Studio 在各个阶段能够帮助完成什么工作。

## 18.2　创建项目

安装 Visual Studio 2017 之后，会希望开始自己的第一个项目。使用 Visual Studio，很少会启动一个空白文件，然后添加 C#代码。但在本书前面的章节中，一直是按这种方式做的(当然，如果想从头开始编写代码，或者打算创建一个包含多个项目的解决方案，则可以选择一个空白解决方案的项目模板)。

在此，告诉 Visual Studio 想创建什么类型的项目，它会生成文件和 C#代码，为该类型的项目提供一个框架。之后在这个基础上继续添加代码即可。例如，如果想创建一个 Windows 桌面应用程序(一个 WPF 应用程序)，Visual Studio 将生成一个 XAML 文件和一个包含 C#源代码的文件，它创建了一个基本的窗体。这个窗体能够与 Windows 通信和接收事件。它能够最大化、最小化或调整大小；用户需要做的仅是添加控件和想要的功能。如果应用程序是一个命令行实用程序(一个控制台应用程序)，Visual Studio 将提供一个基本的名称空间、一个类和一个 Main 方法，用户可以从这里开始。

最后一点也很重要：在创建项目时，Visual Studio 会根据项目是编译为命令行应用程序、库还是 WPF 应用程序，为 C#编译器设置编译选项。它还会告诉编译器，应用程序需要引用哪些基类库和 NuGet 包。例如，WPF GUI 应用程序需要引用许多与 WPF 相关的库，控制台应用程序则不需要引用这些库。当然，在编辑代码的过程中，可以根据需要修改这些设置。

第一次启动 Visual Studio 时，IDE 会包含一些菜单、一个工具栏以及一个包含入门信息、操作方法视频和最新新闻的页面，如图 18-5 所示。起始页包含指向有用网站和一些实际文章的链接，可以打开现有项目或者新建项目。

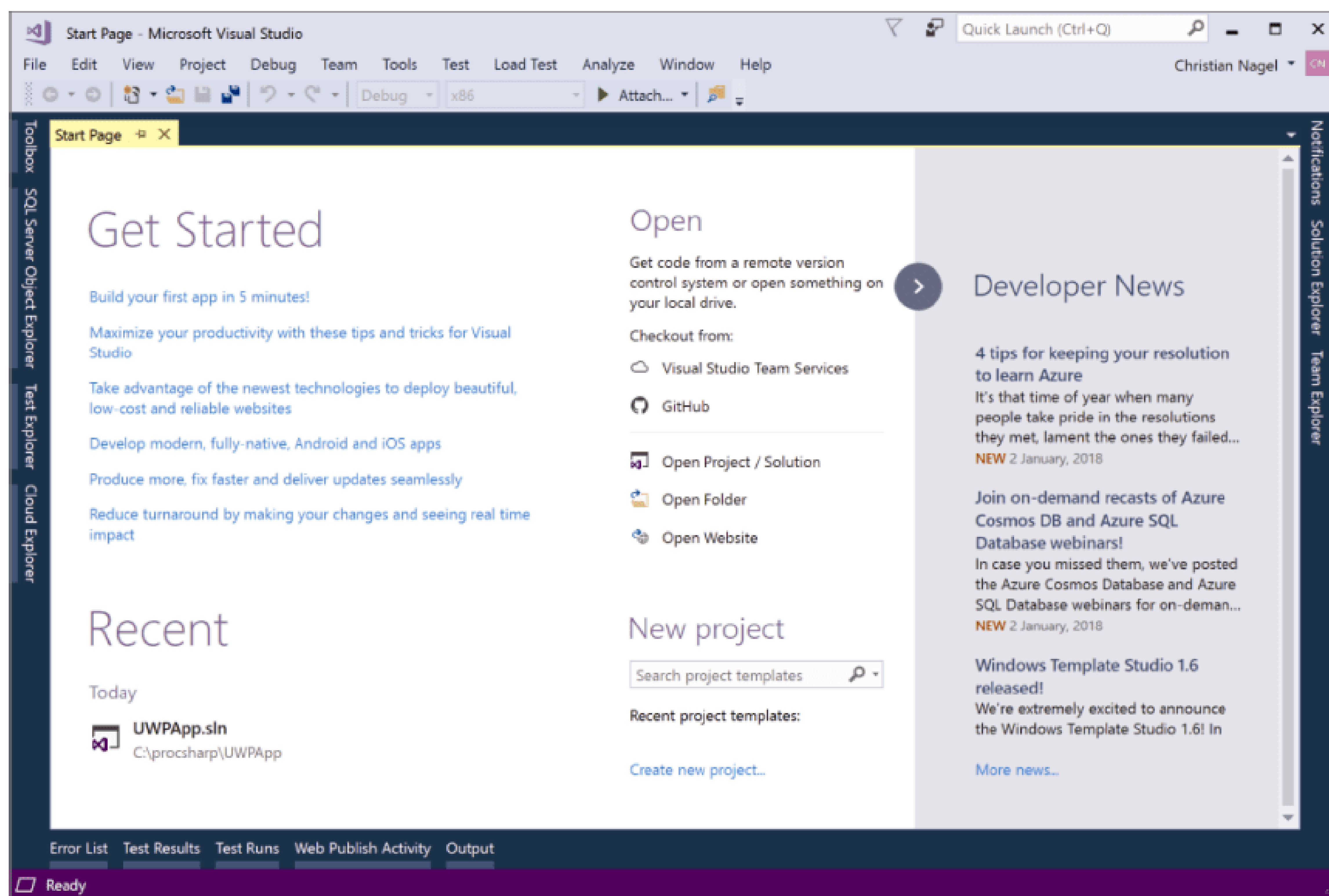



图 18-5

图 18-5 是使用 Visual Studio 2017 后显示的起始页，其中包含最近编辑过的项目列表。单击其中的某个项目就可以打开该项目。

### 18.2.1　面向多个版本的.NET

Visual Studio 2017 允许设置想用于工作的.NET Framework 版本。当打开 New Project 对话框时，如图 18-6 所示，对话框顶部的一个下拉列表显示了可用的选项。

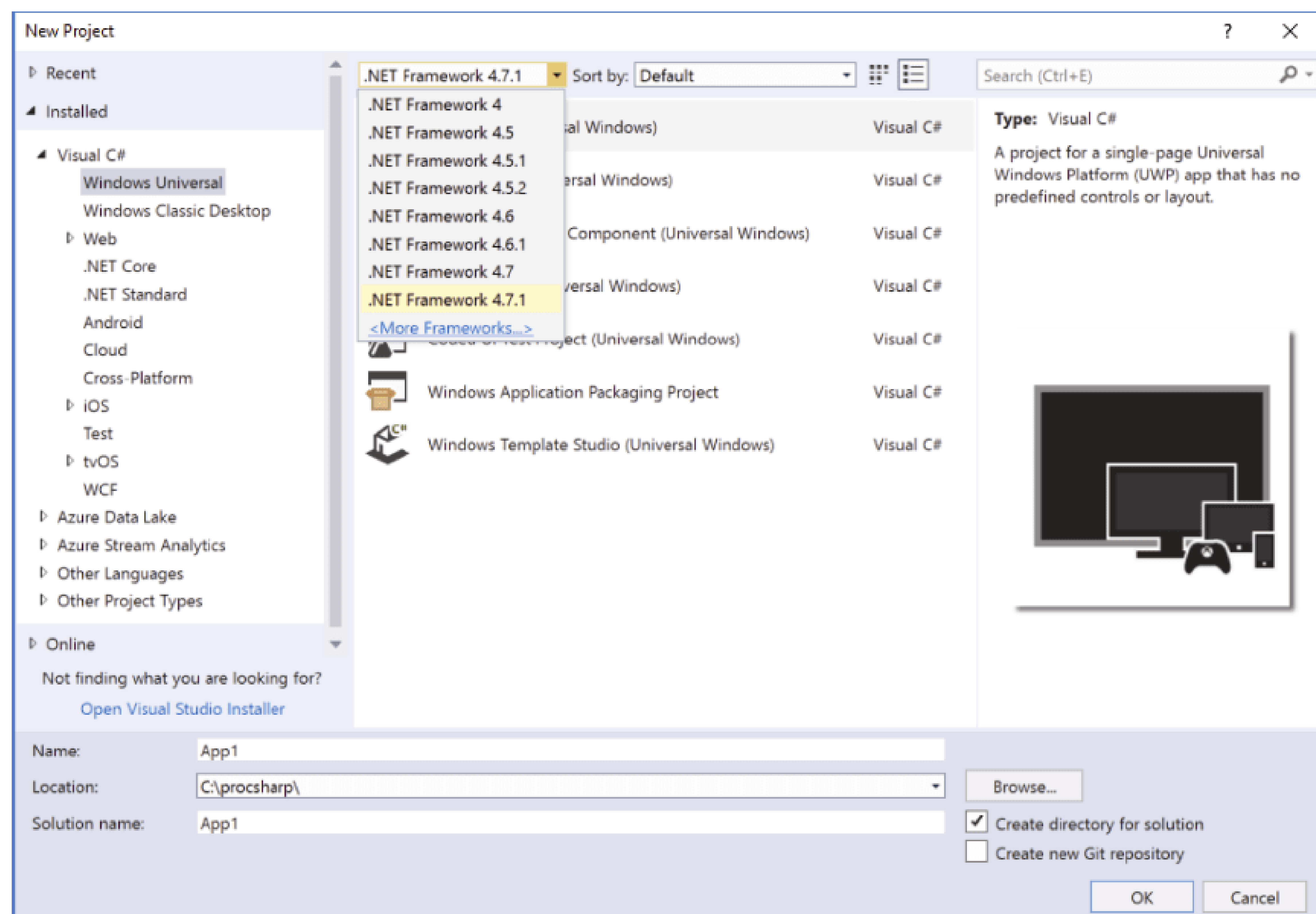



图 18-6

在这种情况下，该下拉列表允许设置的.NET Framework 版本有 4.0、4.5、4.5.1、4.5.2、4.6 和 4.6.1、4.7 和 4.7.1。但是，对于许多应用程序类型，此选项不适用。如果创建.NET Core 应用程序，或者 Windows 通用应用程序，选择什么版本并不重要。不过，可以稍后更改面向的.NET Core 版本或 Windows 运行库版本。

如果想改变.NET Core 应用程序正在使用的框架版本，只需要在 Solution Explorer 中右击项目并选择 Project Properties，选择 Application 选项卡，从 Target Framework 列表中选择.NET Core 版本(参见图 18-7)。

这和 Windows 应用程序没什么不同。也是在 Solution Explorer 中右击项目，并选择 Project Properties，选择 Application 选项卡，现在可以选择目标和最小的构建版本，如图 18-8 所示。

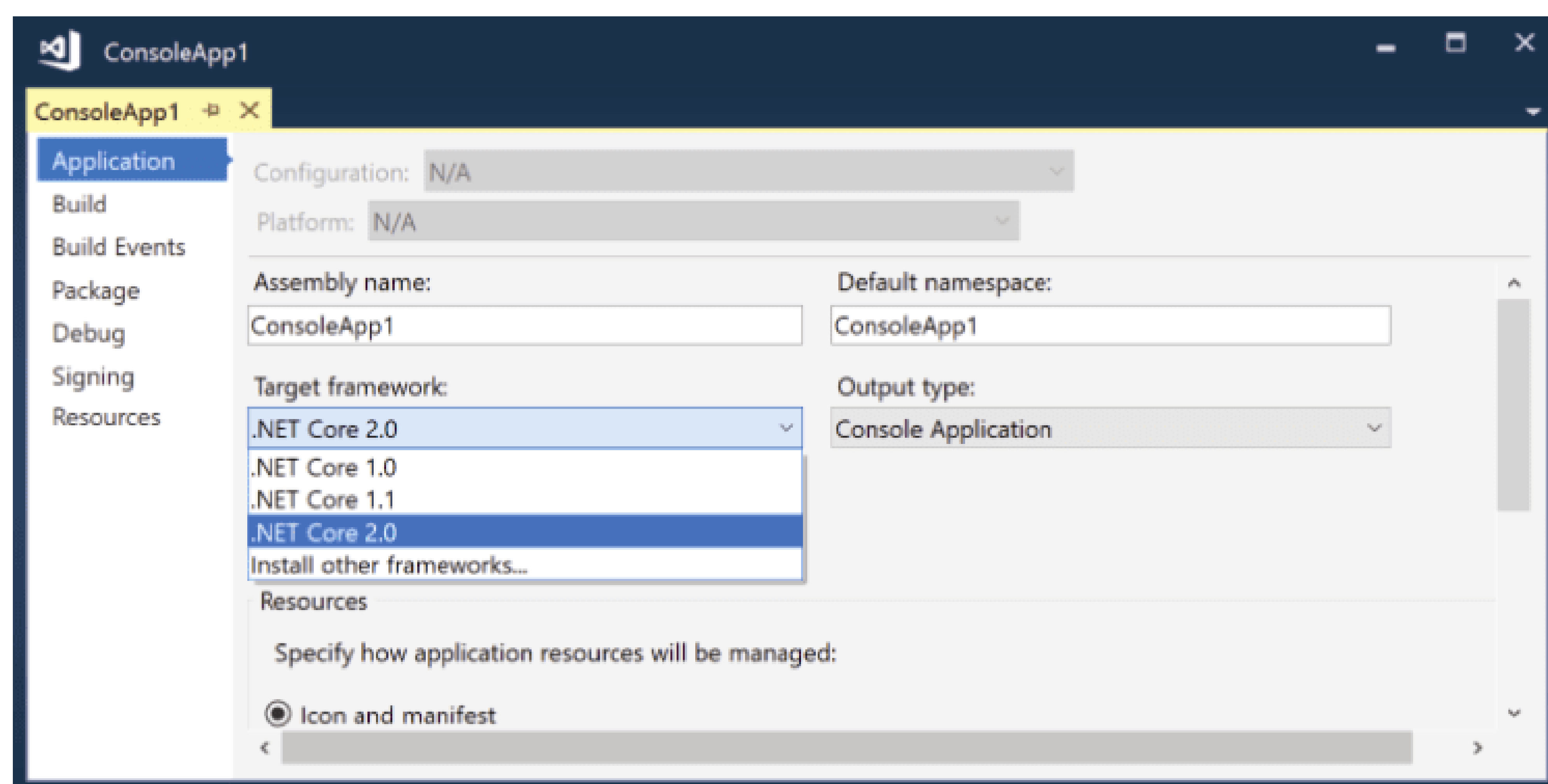



图 18-7

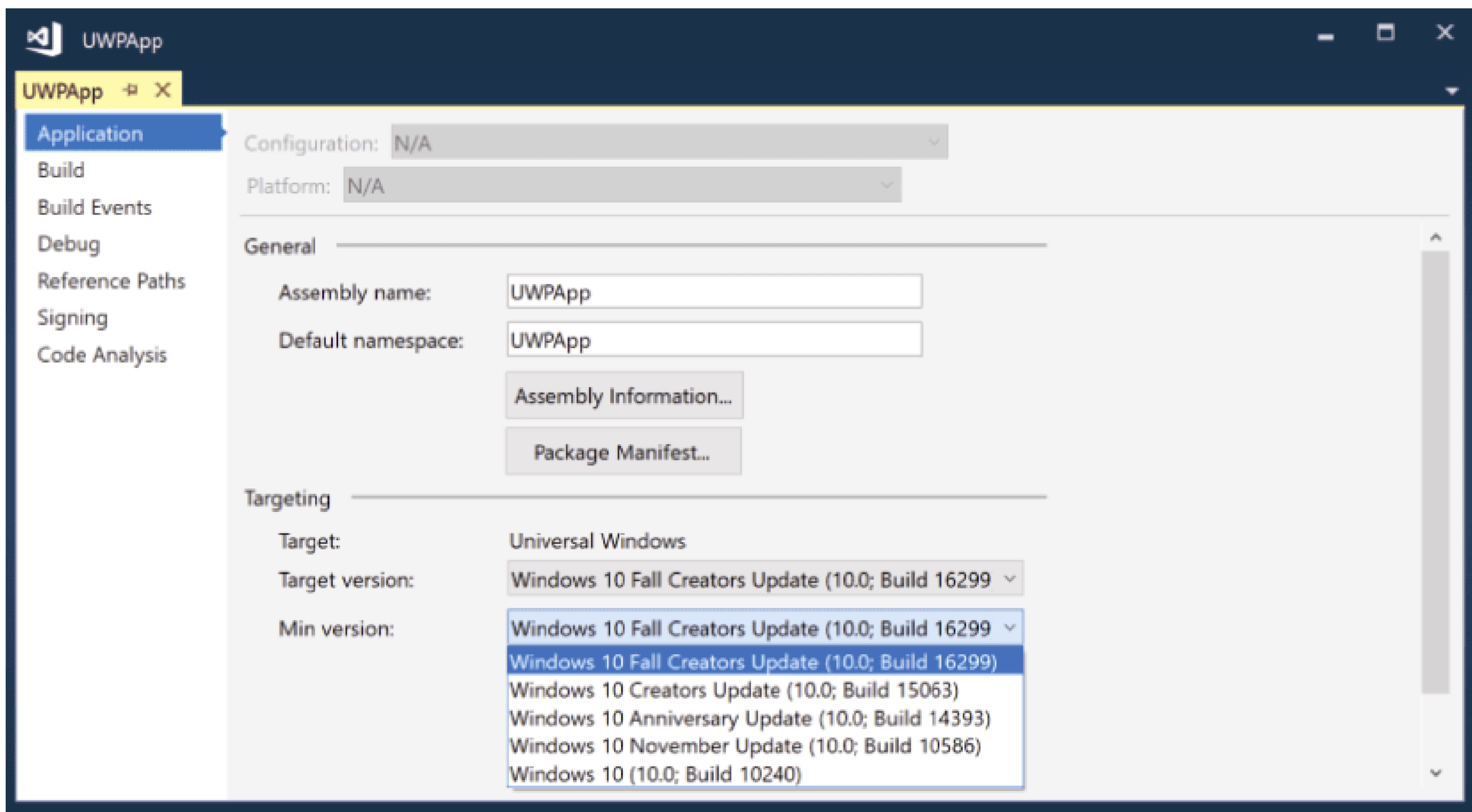



图 18-8

## 18.2.2　选择项目类型

要新建一个项目，从 Visual Studio 菜单中选择 File | New Project。New Project 对话框如图 18-9 所示，通过该对话框可以大致了解能够创建的不同项目。

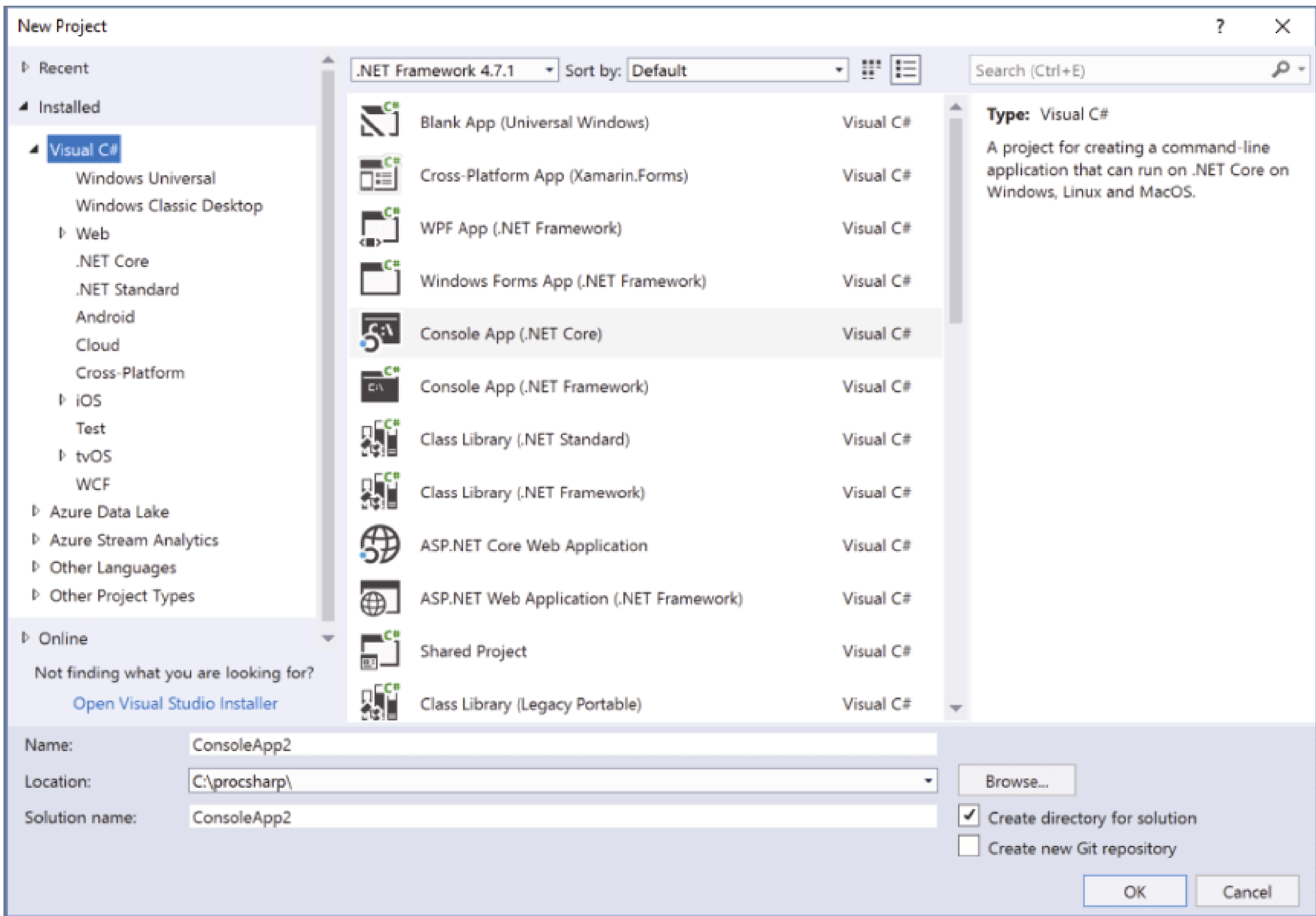



图 18-9

使用这个对话框，可以有效地选择希望 Visual Studio 生成的初始框架文件和代码、希望 Visual Studio 生成的代码、要用于创建项目的编程语言以及应用程序的不同类别。

表 18-1~表 18-3 描述了与本书相关的 Visual C#项目下最重要的选项。这里没有介绍你可能仍然需要的旧项目模板，对于这些模板，应该查阅本书的旧版本。

### 1. 使用 Windows Universal 项目模板

表 18-1 列出了用于 Universal Windows Platform 的模板。这些模板可用于 Windows 10 和 Windows 8.1，但需要 Windows 10 系统来测试应用程序。这些模板用于创建应用程序，使用任何设备系列运行在 Windows 10 上，例如电脑、X-Box、IoT 设备等。

表 18-1

| 项目模板名称 | 项目模板描述 |
|---|---|
| 空白应用程序(Universal Windows) | 一个使用 XAML 的空白 Universal Windows 应用程序，没有样式和其他基类 |
| 类库(Universal Windows) | 一个.NET 类库，其他用.NET 编写的 Windows Store 应用程序可以调用它。在这个库中可以使用 Windows 运行库的 API |
| Windows 运行库组件(Universal Windows) | 一个 Windows Runtime 类库，其他用不同编程语言(C#、C++、JavaScript)开发的 Windows Store 应用程序可以调用它 |
| 单元测试应用程序(Universal Windows) | 一个包含 Universal Windows Platform 应用程序的单元测试的库 |
| 编码的 UI 测试项目(Universal Windows) | 这个项目定义了编码的 UI 测试，用于 Windows 应用程序 |
| Windows 应用程序打包项目 | WPF 或 Windows Forms 项目。可以构建一个 Windows 10 安装包，并将该应用程序与现代 Windows 10 代码混合使用 |

**注意：**

对于 Windows 10，通用应用程序的默认模板数已削减。为了创建 Windows 8 的 Windows Store 应用程序，Visual Studio 提供了更多的项目模板，来预定义基于网格、基于分隔板或者基于 Hub 的应用程序。对于 Windows 10，只有空模板可用。可以从空的模板开始，或考虑使 Windows Template Studio 作为开始。一旦安装了微软的 Template Visual Studio 扩展，Windows Template Studio 模板就可用，其命令是 Tools | Extensions and Updates。

### 2. 使用.NET Core 项目模板

Visual Studio 2017 中的有趣改进是.NET Core 项目模板。最初，只有表 18-2 中的 5 个选项。

表 18-2

| 项目模板名称 | 项目模板描述 |
|---|---|
| 控制台应用程序(.NET Core) | 一个带有.NET Core 的控制台应用。这是在为前几章创建代码时主要使用的模板 |
| 类库(.NET Core) | 可以与.NET Core 应用程序一起使用的类库。如果想在.NET Core、Universal Apps 和 Xamarin 之间共享库，请不要使用此模板。而是寻找标准库。需要使用这个库创建.NET 标准没有提供的特殊.NET Core 功能 |
| 单元测试项目(.NET Core) | 一个单元测试项目，使用 MSTest 测试.NET Core 项目、.NET 标准项目和库 |
| xUnit 测试项目(.NET Core) | 一个单元测试项目，使用 xUnit.测试.NET Core 项目、.NET 标准项目和库 |
| ASP.NET Core Web 应用程序 | ASP.NET Core Web 应用程序，它可以是把 HTML 代码返回到客户端的网站，也可以是运行 JSON 或 XML 的服务。在选择这个项目模板后，可用的选择在表 18-3 中描述 |

在选择 ASP.NET Core Web 应用程序模板后，就可以选择一些预先配置的模板，如图 18-10 所示。使用顶部的组合框来选择.NET Core 或.NET Framework。ASP.NET Core 也可以在.NET Framework 上运行，而不仅仅是在.NET Core 上运行。然后可以选择 ASP.NET Core 版本号，这取决于已安装的 SDK。只有需要使用仅与.NET Framework 一起运行的旧库时，.NET Framework 选项才有用，否则保留.NET Core 选项。如果选择.NET Core 和 ASP.NET Core 2.0，就将看到如图 18-10 所示的屏幕。这些模板将在表 18-3 中描述。

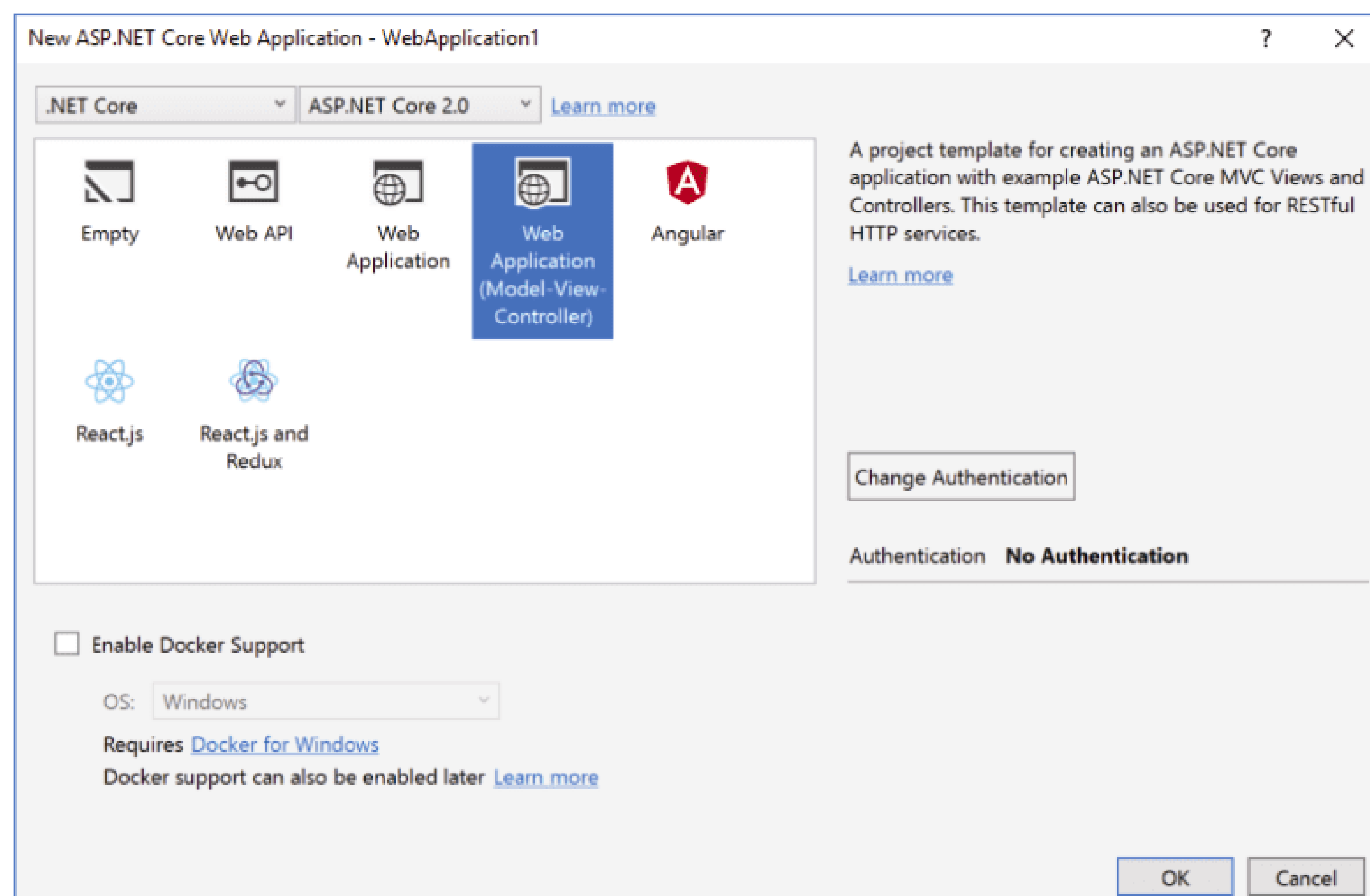



图 18-10

表 18-3

| 项目模板名称 | 项目模板描述 |
|---|---|
| 空白应用程序 | 一个 ASP.NET Core Web 应用程序。在选择这个模板时，不会得到一个完全空的项目，而是使用.NET Core 创建一个基本的 Web 应用程序。这个模板是在第 30 和 31 章中开始使用的。在这两章会学习需要添加什么内容 |
| Web API | 使用 ASP.NET Core 提供 Web API 的服务。Web API 模板很容易创建 RESTful 服务。这个项目参见第 32 章 |
| Web 应用程序 | 具有 Razor 页面的 Web 应用程序。这是 ASP.NET Core 2.0 的一个新选项，在第 31 章中介绍 |
| MVC | 使用 ASP.NET Core MVC 的 Web 应用程序。这个模板使用完全“模型-视图-控制器”模式。它可以用于创建 Web 应用程序，在第 31 章中介绍 |
| Angular | 使用 Angular 脚本库和 ASP.NET Core 为后端服务创建单页应用程序(Single Page Application，SPA)的 Web 应用程序 |
| React.js | 为后端服务使用 React 和 ASP.NET Core 的 Web 应用程序。React.js 是另一种 SPA 技术 |
| React.js 和 Redux | 该 Web 应用程序给客户端使用 React.js 和 Redux，为后端服务使用 ASP.NET Core,。这一次，除了 React.js 之外，还使用了 Redux 库 |

### 3. 使用.NET 标准模板

这个类别只包含一个模板，但它非常重要，所以这里介绍它。可以创建一个类库(.NET 标准)。从现在开始，这是要创建的首选类库。这个库可以在.NET Framework、.NET Core、通用应用程序、Xamarin 和更多技术之间共享。只需要注意创建这个库后所选择的标准的版本。

.NET 标准库详见第 19 章。

**注意：**
*.NET 标准库取代了可移植库。现在，可移植的库在 Visual Studio 中被列为旧库。*

这不是一个完整的 Visual Studio 2017 项目模板列表，但它列出了一些最常用的模板。

# 18.3 浏览并编写项目

本节着眼于 Visual Studio 提供用于帮助在项目中添加和浏览代码的功能。学习如何使用 Solution Explorer 浏览文件和代码，使用编辑器的 IntelliSense 和代码片段等功能浏览其他窗口，如 Properties(属性)窗口和 Document Outline(文档大纲)。

## 18.3.1 Solution Explorer

在创建项目[例如，前面章节最常用的控制台应用程序(.NET Core)]之后，要用到的最重要的工具除了代码编辑器，就是 Solution Explorer。使用这个工具可以浏览项目的所有文件和项，查看所有的类和类成员。

**注意：**

在 Visual Studio 中运行控制台应用程序时，有一个常见的误解，即需要在 Main()方法的最后一行添加一个 Console.ReadLine 方法来保持控制台窗口打开。事实并非如此，通过命令 Debug | Start without Debugging(或按 Ctrl+F5 组合键)可以启动应用程序，而不必通过命令 Debug | Start Debugging(或按 F5 键)来开启。这样可以保持窗口打开直到按下某个键。使用 F5 键开启应用程序也是有意义的，如果设置了断点，Visual Studio 就会在断点处挂起。

### 1. 使用项目和解决方案

Solution Explorer 会显示项目和解决方案。理解它们之间的区别是很重要的：

- 项目是一个包含所有源代码文件和资源文件的集合，它们将编译成一个程序集，在某些情况下也可能编译为一个模块。例如，项目可能是一个类库或一个 Windows GUI 应用程序。
- 解决方案是一个包含所有项目的集合，它们组合成一个特定的软件包(应用程序)。

要理解这个区别，可以考虑当发布一个包括多个程序集的项目时会发生什么。例如，可能有用户界面、自定义控件和作为应用程序一部分的库的其他组件。甚至可能为管理员提供不同的用户界面和通过网络调用的服务。应用程序的每一部分可能包含在单独的程序集中，因此 Visual Studio 会认为它们是单独的项目。而且很有可能并行编码这些项目，并将它们彼此结合。因此，在 Visual Studio 中把这些项目当作一个单位来编辑是非常有用的。Visual Studio 允许把所有相关的项目构成一个解决方案，并且当作一个单位来处理，Visual Studio 会读取该单位并允许在该单位上进行工作。

到目前为止，本章已经零散地讨论创建一个控制台项目。在这个例子中，Visual Studio 实际上已经创建一个解决方案，只不过它仅包含一个项目而已。可以在 Solution Explorer 中看到这样的场景(如图 18-11 所示)，它包含一个树型结构，用于定义该解决方案。

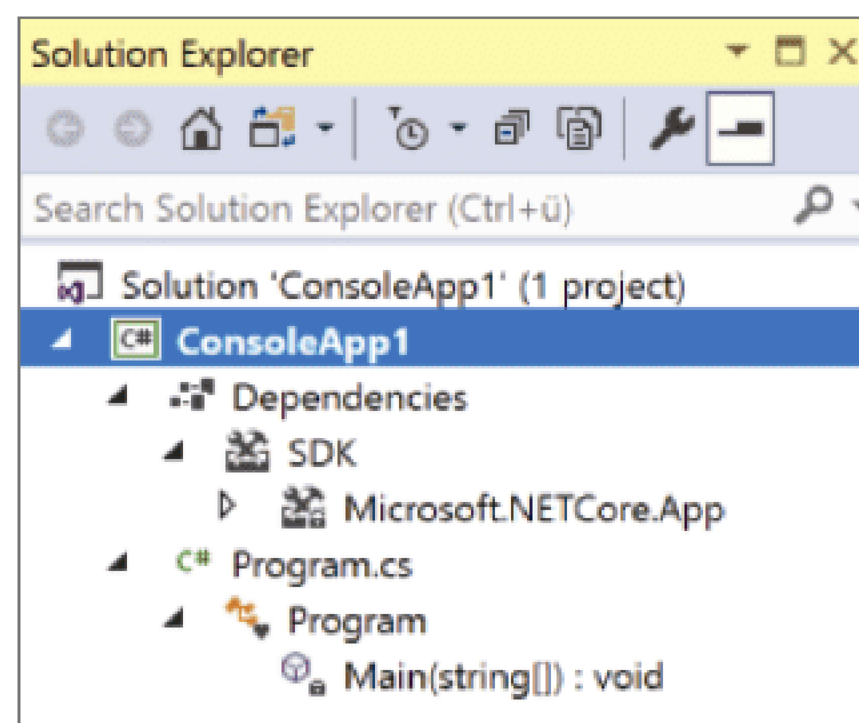



图 18-11

在这个例子中，项目包含了源文件 Program.cs，以及项目配置文件 project.json(允许定义项目描述、版本和依赖项)。在 Solution Explorer 中无法清楚地看到项目文件。只需要选择项目(图 18-11 中的 ConsoleApp1)，然后在上下文菜单中选择 Edit ConsoleApp1.csproj(单击键盘上的菜单按钮或右击)。有了.NET Core 项目，就可以这么做，而不需要卸载解决方案。使用其他项目类型(例如，Universal Windows Apps)时，在直接从 Visual Studio 中编辑项目文件之前，需要首先卸载解决方案。

Solution Explorer 也显示了项目引用的 NuGet 包和程序集。在 Solution Explorer 中展开 Dependencies 文件夹就可以看到这些信息。

**注意：**

对于较旧的项目类型，会看到 References 文件夹，而不是 Dependencies 文件夹。

如果在 Visual Studio 中没有改变任何默认设置，在屏幕右上方就可以找到 Solution Explorer。如果找不到它，则可以进入 View 菜单并选择 Solution Explorer。

解决方案是用一个扩展名为.sln 的文件来描述的，在这个示例中，它是 ConsoleApp1.sln。解决方案文件是一个文本文件，它包含解决方案中包含的所有项目的信息，以及可用于所有包含项目的全局项。

**显示隐藏文件**

默认情况下，Solution Explorer 隐藏了一些文件。单击 Solution Explorer 工具栏中的 Show All Files 按钮，可以显示所有隐藏的文件。例如，bin 和 obj 子文件夹存放了编译的文件和中间文件。obj 子文件夹存放各种临时的文件或中间文件；bin 子文件夹存放已编译的程序集。

### 2. 将项目添加到解决方案中

下面各节将介绍 Visual Studio 如何处理 Windows 桌面应用程序和控制台应用程序。最终会创建一个名为 SimpleApp 的 Windows 项目，将它添加到当前的解决方案 ConsoleApp1 中。

**注意：**

创建 SimpleApp 项目，得到的解决方案将包含一个 UWP 应用程序和一个控制台应用程序。这种情况并不多见，更有可能的是解决方案包含一个应用程序和许多类库。这么做只是为了展示更多的代码。不过，有时需要创建这样的解决方案，例如，编写一个既可以运行为 UWP 应用程序、又可以运行为命令行实用工具的实用程序。

创建新项目的方式有几种。一种方式是在 File 菜单中选择 New | Project(前面就是这么做的)，或者在 File 菜单中选择 Add | New Project。选择 New Project 命令将打开熟悉的 Add New Project 对话框，如图 18-12 所示。不过，此时 Visual Studio 会在已有 ConsoleApp1 项目所在的解决方案中创建新项目。

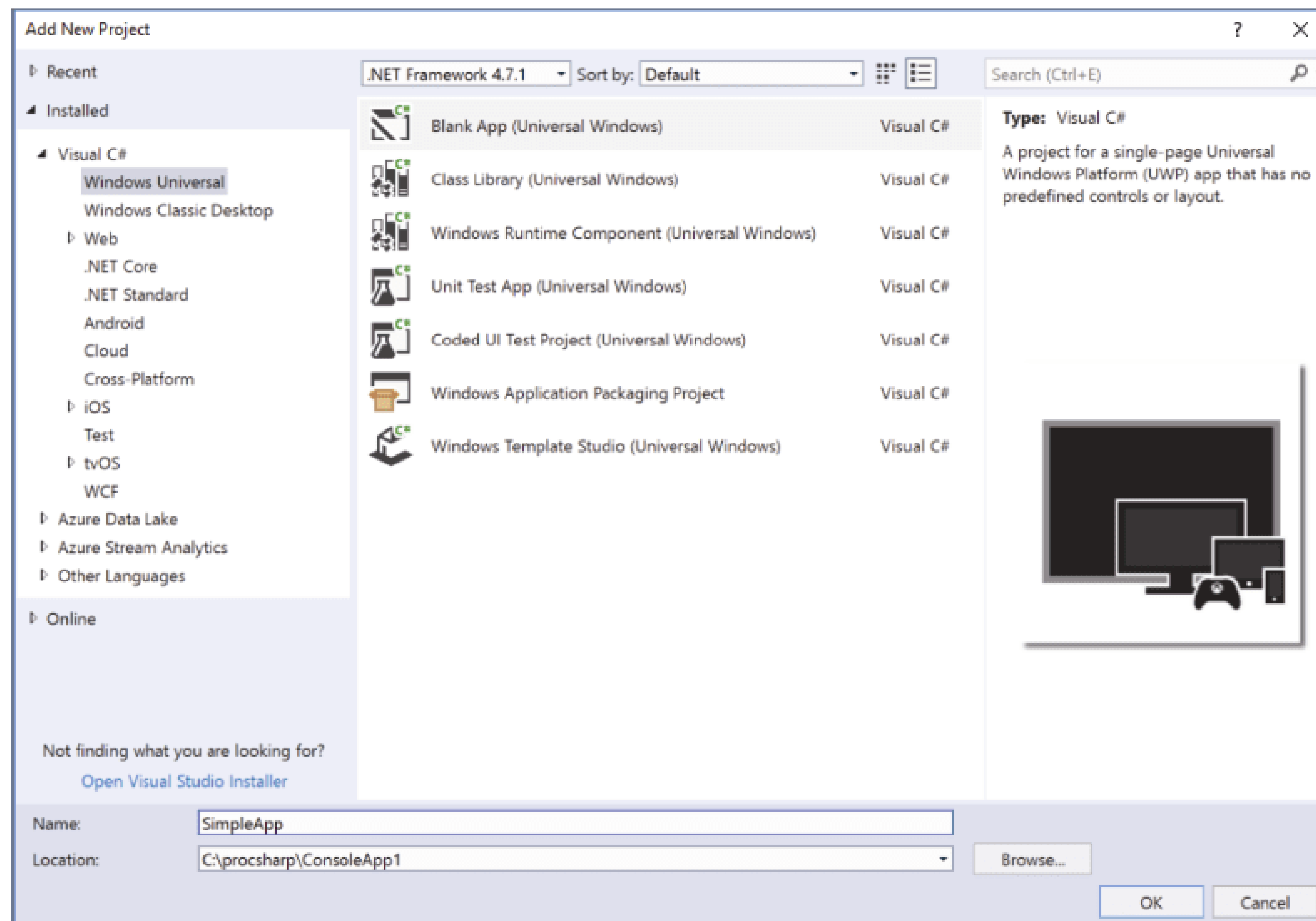



图 18-12

如果选择该选项，就会添加一个新项目，因此 ConsoleApp1 解决方案现在包含一个控制台应用程序和一个空白的应用程序(Universal Windows)。

**注意：**

Visual Studio 支持语言独立性，所以新项目并不一定是 C#项目。将 C#项目、Visual Basic 项目和 C++项目放在同一个解决方案中是完全可行的。但是，本书的主题是 C#，所以创建 C#项目。

对于平台版本，选择安装在系统上的 Target 和 Minimum 版本。可以为 Target 和 Minimum 版本选择最新版本(参见图 18-13)。

当然，这意味着 ConsoleApp1 不再适合作为解决方案的名称。要改变名称，可以右击解决方案的名称，并选择上下文菜单中的 Rename 命令。将新的解决方案命名为 DemoSolution。Solution Explorer 窗口现在如图 18-14 所示。

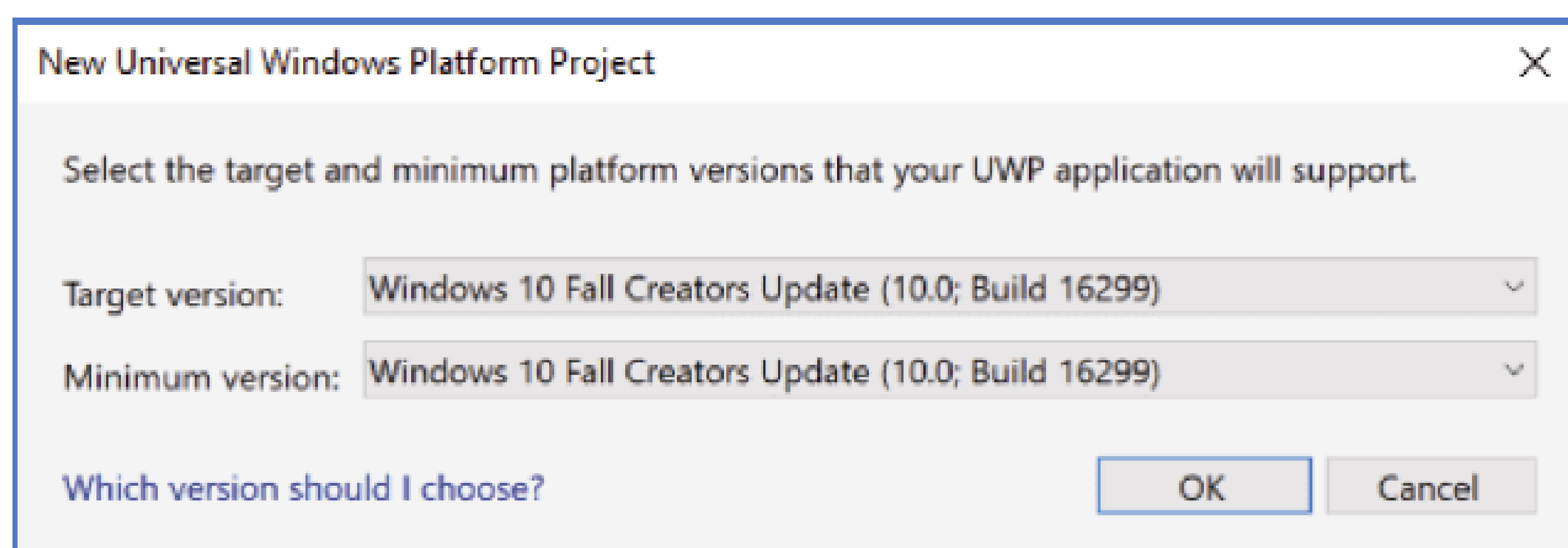



图 18-13

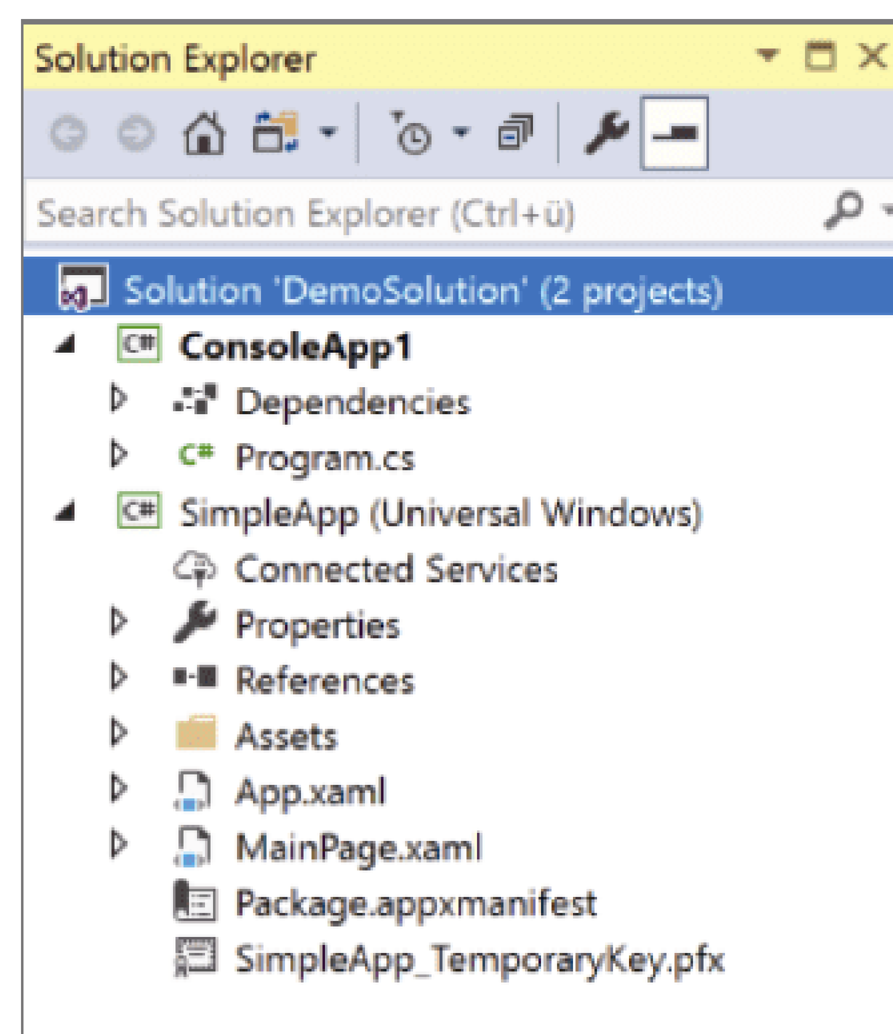



图 18-14

可以看出，Visual Studio自动为新添加的UWP项目引用一些额外的基类，这些基类对于WPF功能非常重要。

注意，在 Windows Explorer 中，解决方案文件的名称已经改为 DemoSolution.sln。通常，如果想重命名任何文件，Solution Explorer 窗口是最合适的选择，因为 Visual Studio 会自动更新它在其他项目文件中的引用。如果只使用 Windows Explorer 重命名文件，可能会破坏解决方案，因为 Visual Studio 不能定位需要读入 IDE 的所有文件。因此需要手动编辑项目和解决方案文件来更新文件引用。

### 3. 设置启动项目

请记住，如果一个解决方案有多个项目，就需要配置哪个项目作为启动项目来运行。也可以配置多个同时启动的项目。这有多种方式。在Solution Explorer中选择一个项目之后，上下文菜单会提供Set as Startup Project选项，它允许一次设置一个启动项目。也可以使用上下文菜单中的Debug | Start new instance命令，在一个项目后启动另一个项目。要同时启动多个项目，右击Solution Explorer中的解决方案，并选择上下文菜单中的Set Startup Projects，打开如图18-15所示的对话框。当选择Multiple Startup Projects之后，可以定义启动哪些项目。

### 4. 浏览类型和成员

当 Visual Studio 初次创建 UWP 应用程序时，该应用程序比控制台应用程序要多包含一些初始代码。这是因为创建窗口是一个较复杂的过程。第 34 章详细讨论 UWP 应用程序的代码。现在，查看 MainPage.xaml 中的 XAML 代码，和 MainPage.xaml.cs 中的 C#代码。这里也有一些隐藏的 C#代码。遍历 Solution Explorer 中的树状结构，在 MainPage.xaml.cs 的下面可以找到 MainPage 类。Solution Explorer 在该文件中显示了所有代码文件中的类型。在 MainPage 类型中，可以看到类的所有成员。_contentLoaded 是一个布尔类型的字段。单击这个字段，会打开 MainWindow.g.i.cs 文件。这个文件是 MainPage 类的一部分，它由设计器自动生成，包含一些初始化代码。

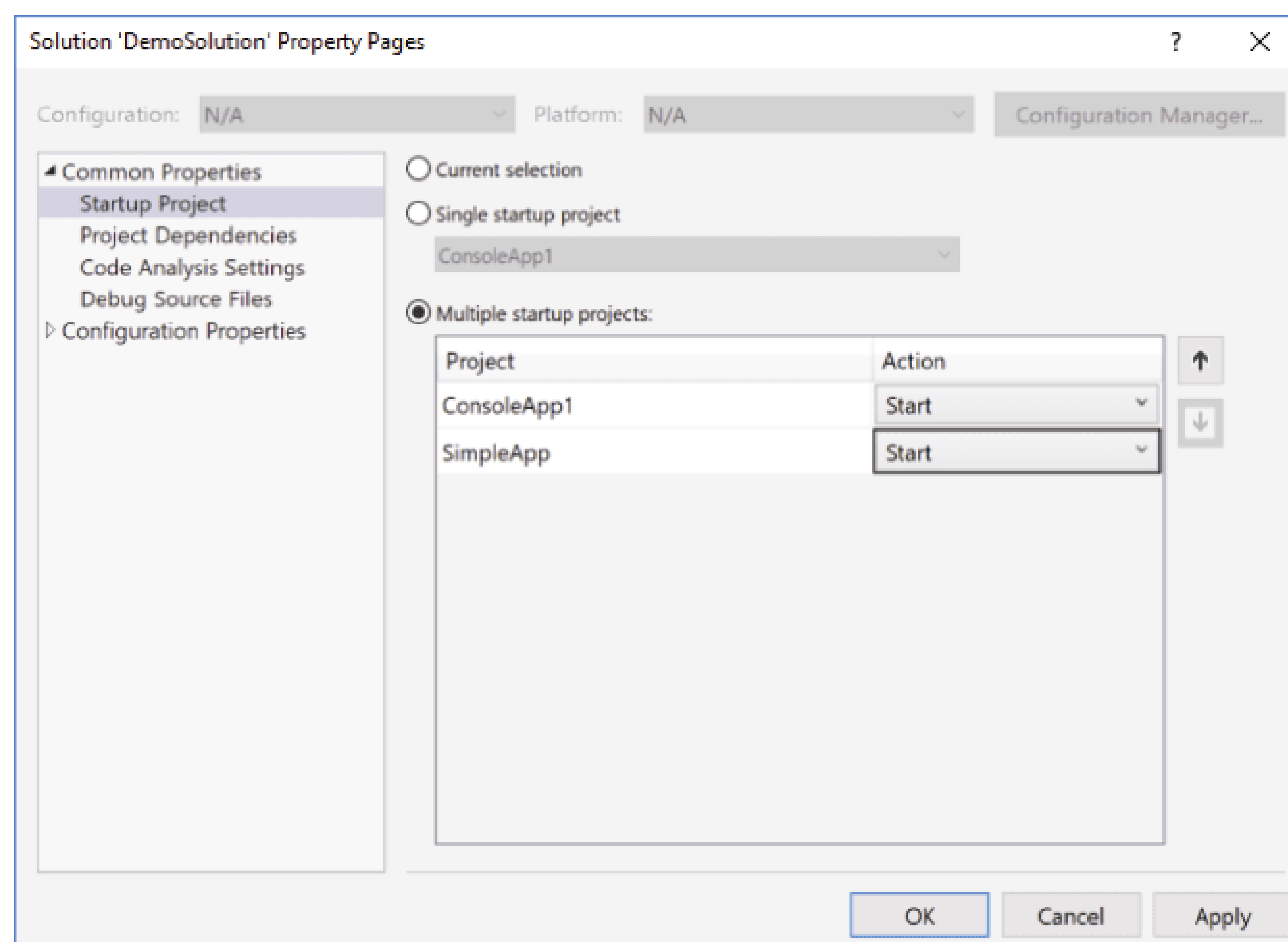



图 18-15

### 5. 预览项

Solution Explorer 提供的一个功能是 Preview Selected Items 按钮。启用这个按钮，在 Solution Explorer 中单击一项，就会打开该项的编辑器，这与往常相同。但如果该项以前没有打开过，编辑器的选项卡流就会在最右端显示新打开的项。现在单击另一项，以前打开的项就会关闭。这大大减少了打开的项数。

在预览项的编辑器选项卡中有 Keep Open 按钮，它会使该项在单击另一项时仍处于打开状态，保持打开的项的选项卡会向左移动。

### 6. 使用作用域

设置作用域可以让用户专注于解决方案的某一特定部分。Solution Explorer 列表显示的项会越来越多。例如，打开一个类型的上下文菜单，就可以从 Base Types 菜单中选择该类型的基类型。这里可以看到完整的类型继承层次结构，如图 18-16 所示。

因为 Solution Explorer 包含的信息量比在一个屏幕中可以轻松查看的信息量要多，所以可以用 New Solution Explorer View 菜单项一次打开多个 Solution Explorer 窗口，并且可以设置作用域来显示一个特定元素。例如，要显示一个项目或一个类，可选择上下文菜单中的 Scope to This 命令。要返回到以前的作用域，可单击 Back 按钮。

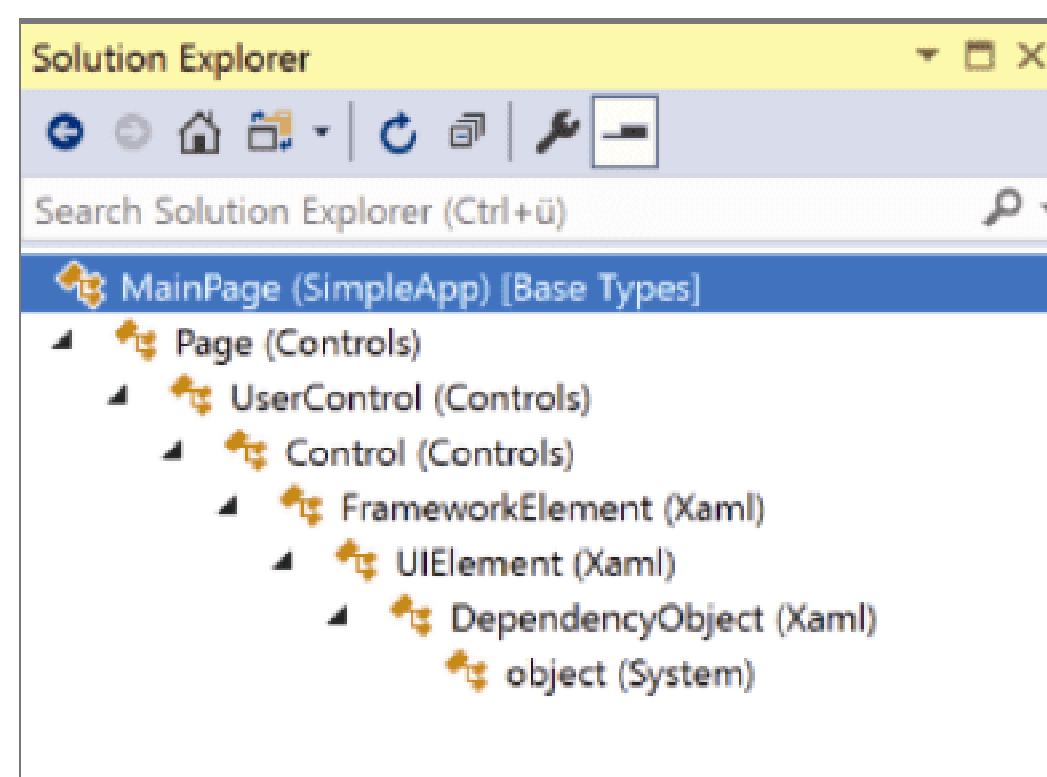



图 18-16

### 7. 将项添加到项目中

在 Solution Explorer 中可以直接将不同的项添加到项目中。选择项目，打开上下文菜单 Add | New Item，打开如图 18-17 所示的对话框。打开这个对话框的另一种方式是，使用主菜单 Project | Add New Item。该对话框有很多不同的类别，例如添加类或接口的代码项、使用 Entity Framework 或其他数据访问技术的数据项等。

### 8. 管理引用和依赖项

使用 Visual Studio 添加引用需要特别关注，因为项目类型有区别。根据项目类型，可以在 Solution Explorer 中看到 Dependencies 或 References。在单击此项时打开上下文菜单，可以看到 Add References 菜单项，它用以打开 Reference Manager。

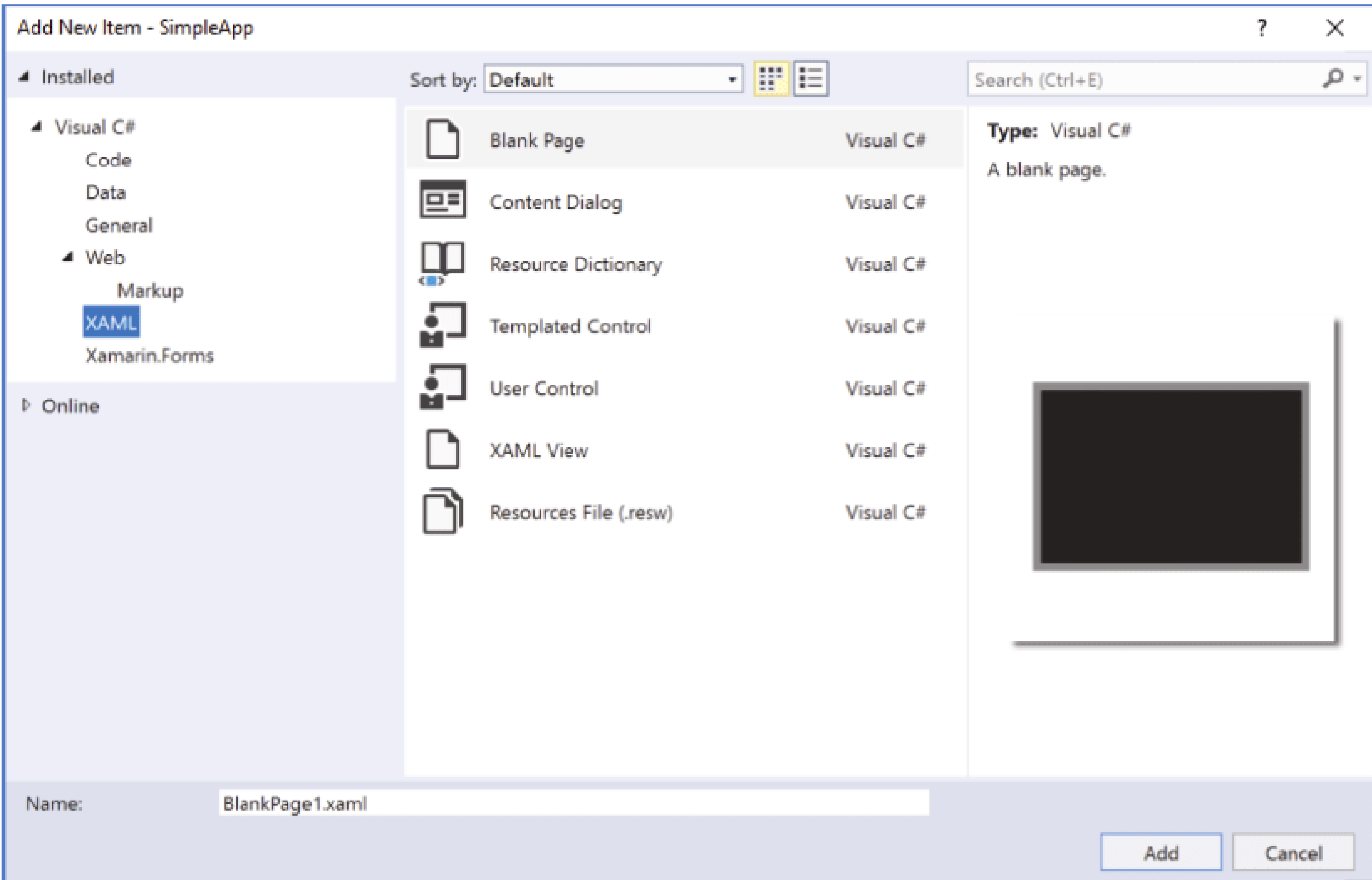



图 18-17

如图 18-18 所示的 Reference Manager 可以向同一解决方案中添加对其他项目的引用，向同一解决方案中添加对共享项目的引用，并浏览程序集。

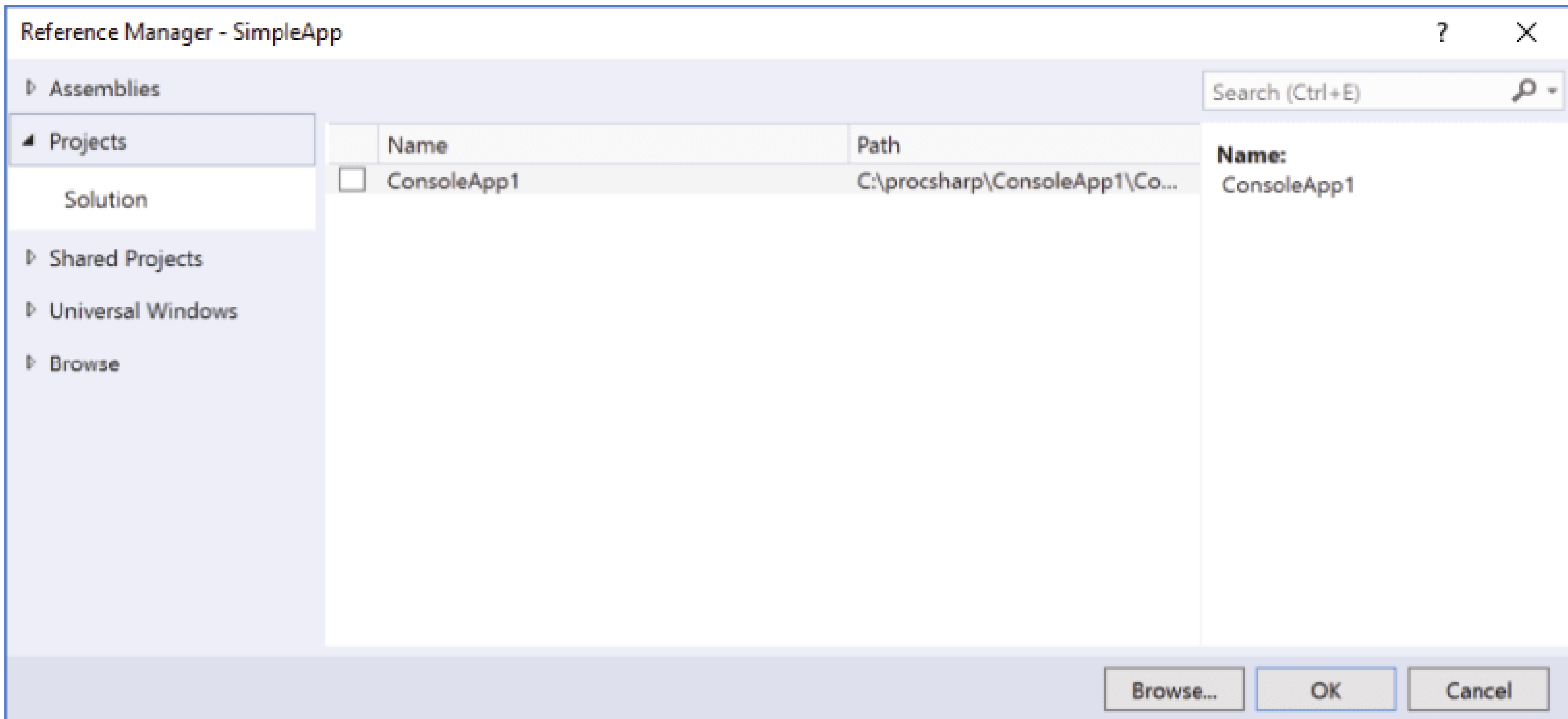



图 18-18

**注意：**
共享项目允许在不同的技术之间共享代码，而不需要创建库。这个特性在第 19 章中介绍。

根据要添加引用的项目类型，Reference Manager 提供了不同的选项。对于.NET Framework 项目，还可以引用共享程序集和 COM 对象。

当创建 Universal Windows Platform 应用程序时，可以引用 Universal Windows Extensions，例如可用于 Windows IoT 或 Windows Mobile 的 API 扩展，如图 18-19 所示。

.NET Core 的所有新功能都可以通过 NuGet 包使用。许多对.NET Framework 的改进也可以通过 NuGet 包使用。可以在 Solution Explorer 的上下文菜单中访问 NuGet 包管理器(请参见图 18-20)，也可以选择 Project | Manage NuGet Packages。可以浏览要安装的新软件包，查看已安装的软件包，更新已安装的软件包。图 18-20 显示了包更新可用的指示。

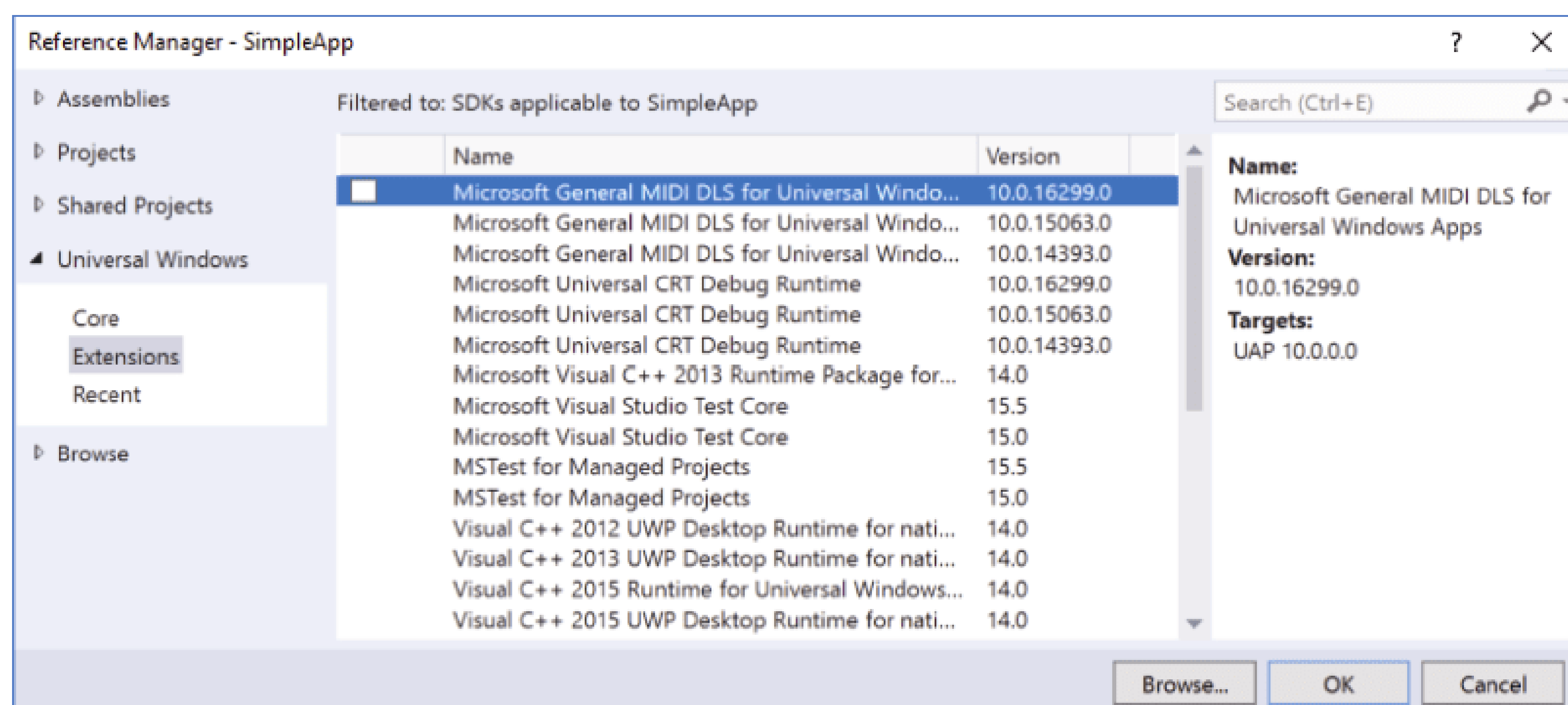



图 18-19

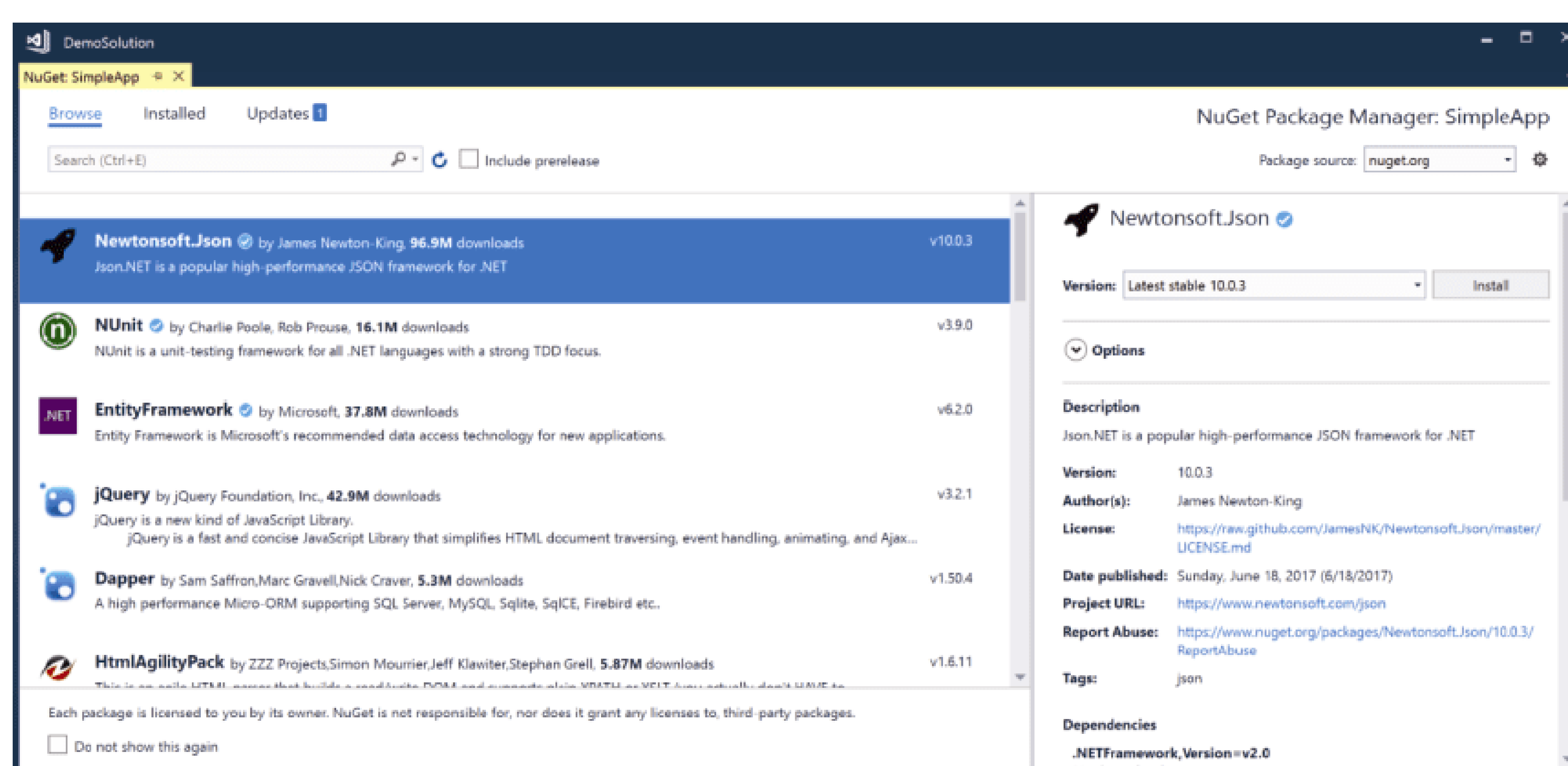



图 18-20

要配置 NuGet 包的来源，可以通过选择 Tools | Options，打开 Options 对话框。在 Options 对话框中选择树视图中的 NuGet Package Manager | Package Sources (见图 18-21)。默认情况下，配置了微软的 NuGet 服务器，也可以配置其他 NuGet 服务器或自己的服务器。在.NET Core 和 ASP.NET Core 1.0 中，微软提供了每日更新的 NuGet 包种子。

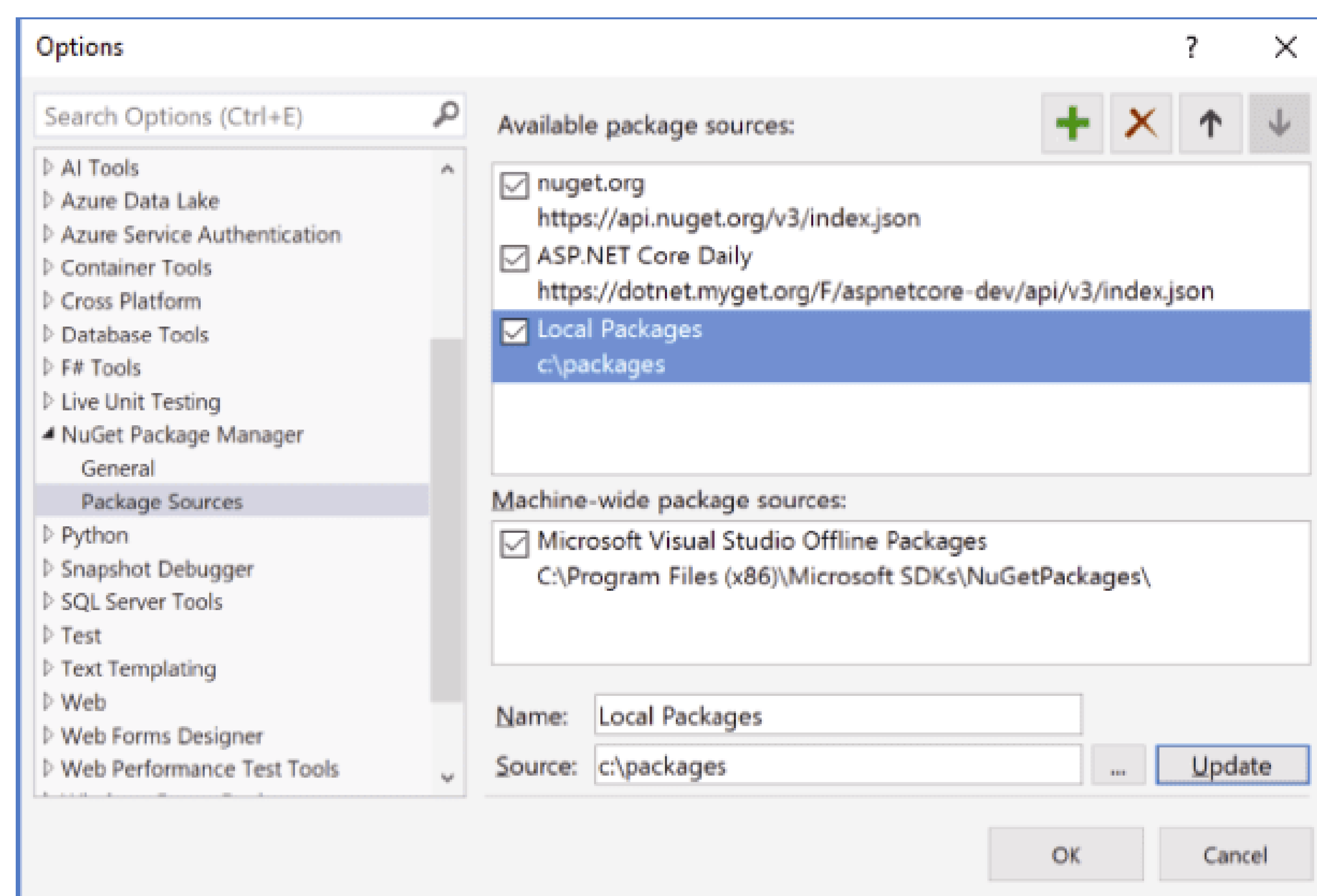



图 18-21

使用 NuGet 包管理器，不仅可以选择包的来源，也可以选择一个过滤器，查看安装的所有包，或者可用的升级包，并搜索服务器上的包。

**注意：**

在 ASP.NET Core 中，JavaScript 库不再在 NuGet 服务器中使用。相反，JavaScript 包管理器，例如 NPM、Bower 等，在 Visual Studio 2017 中获得直接支持。参见第 30 章。

依赖项或引用的另一个选项是 Add Connected Service 菜单。这将打开如图 18-22 所示的对话框，该对话框允许向应用程序轻松添加特定的特性。选择这些包中的一个会添加 NuGet 包，在应用程序中做一些修改，通常打开一个网页来帮助确定下一步需要做什么。可以通过在这个对话框中的 Find More Services...链接来访问更多的连接服务。

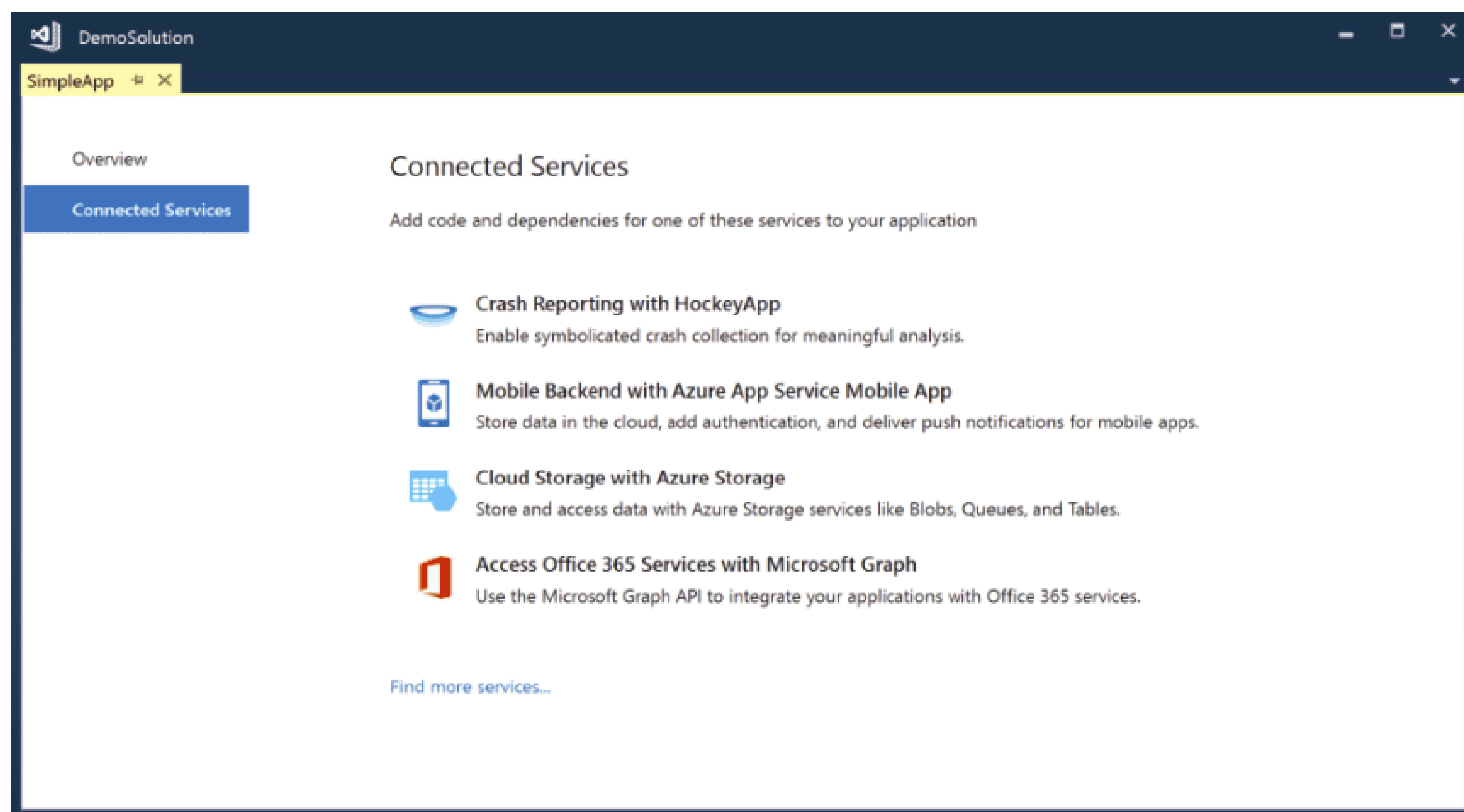



图 18-22

## 18.3.2 使用代码编辑器

Visual Studio 代码编辑器是进行大部分开发工作的地方。在 Visual Studio 中，从默认配置中移除一些工具栏，并移除了菜单栏、工具栏和选项卡标题的边框，从而增加了代码编辑器的可用空间。下面介绍该编辑器中最有用的功能。

### 1. 可折叠的编辑器

Visual Studio 中的一个显著功能是使用可折叠的编辑器作为默认的代码编辑器。图 18-23 是前面生成的控制台应用程序代码。注意窗口左侧的小减号，这些符号所标记的点是编辑器认为新代码块(或文档注释)的开始位置。可以单击这些图标来关闭相应代码块的视图，如同关闭树状控件中的节点，如图 18-24 所示。

这意味着在编辑时可以只关注所需的代码区域，隐藏此刻不感兴趣的代码。如果不喜欢编辑器折叠代码的方式，可以用 C#预处理器指令#region 和#endregion 来指定要折叠的代码块。例如，要折叠 Main()方法中的代码，可以添加如图 18-25 所示的代码。

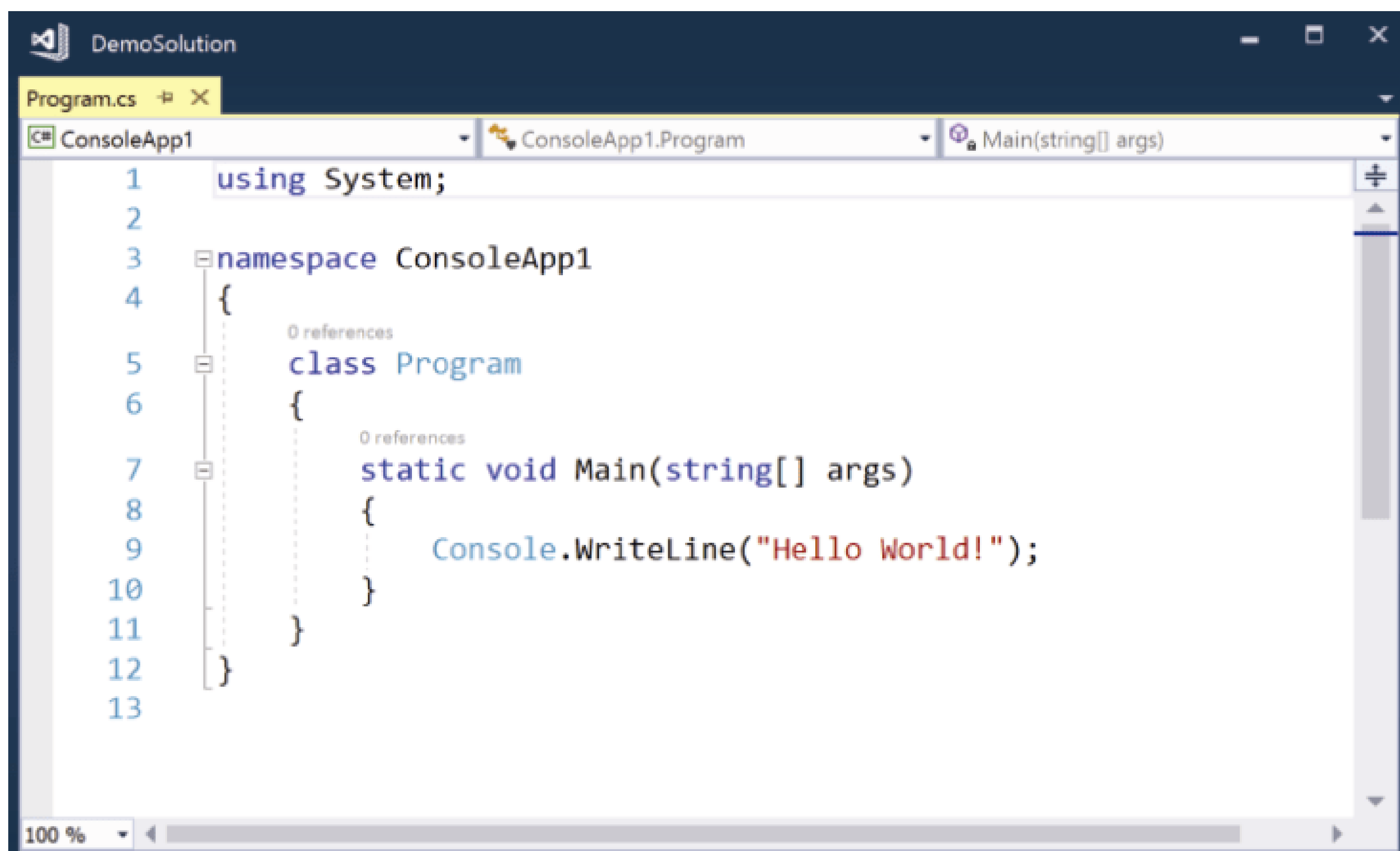



图 18-23

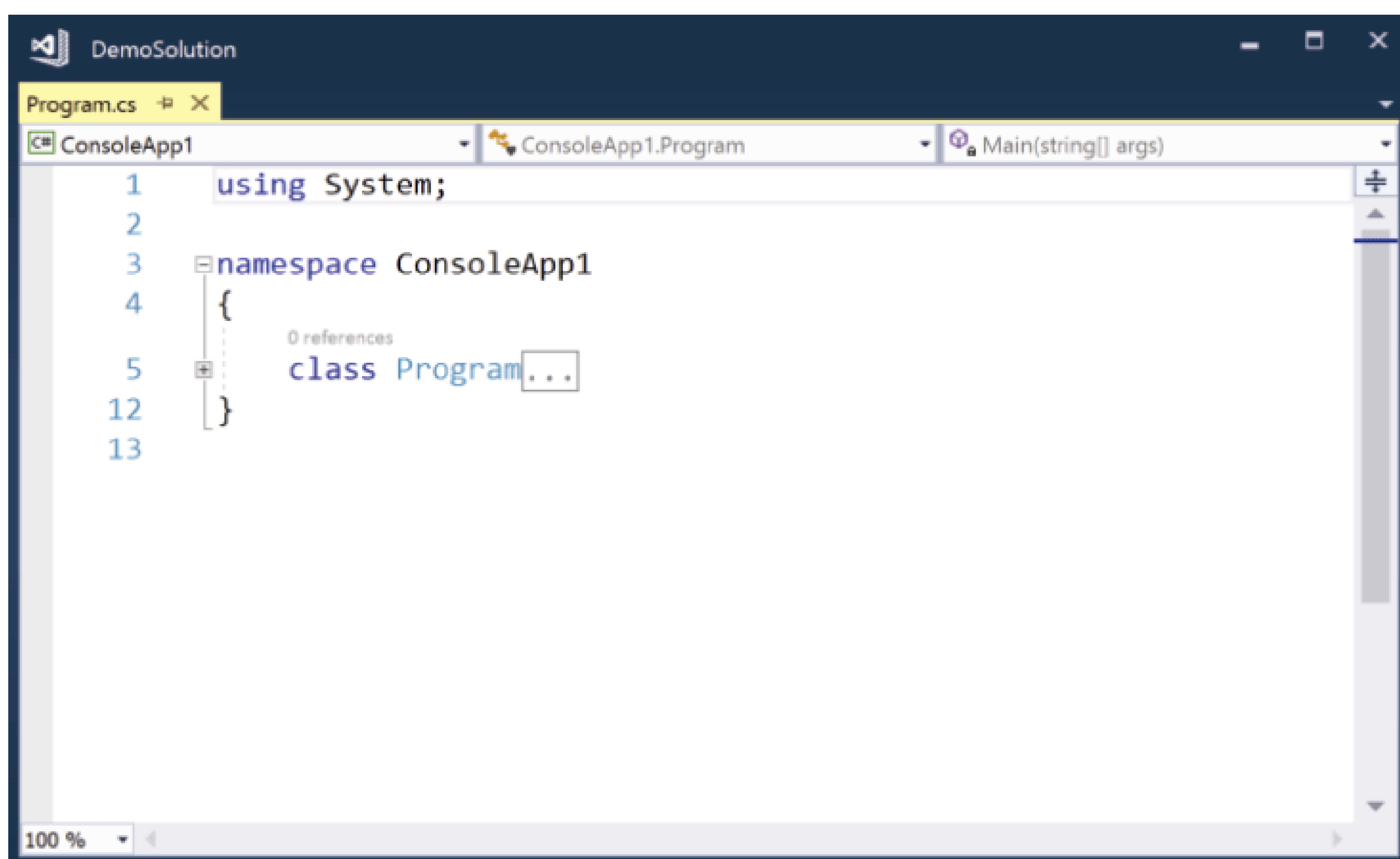



图 18-24

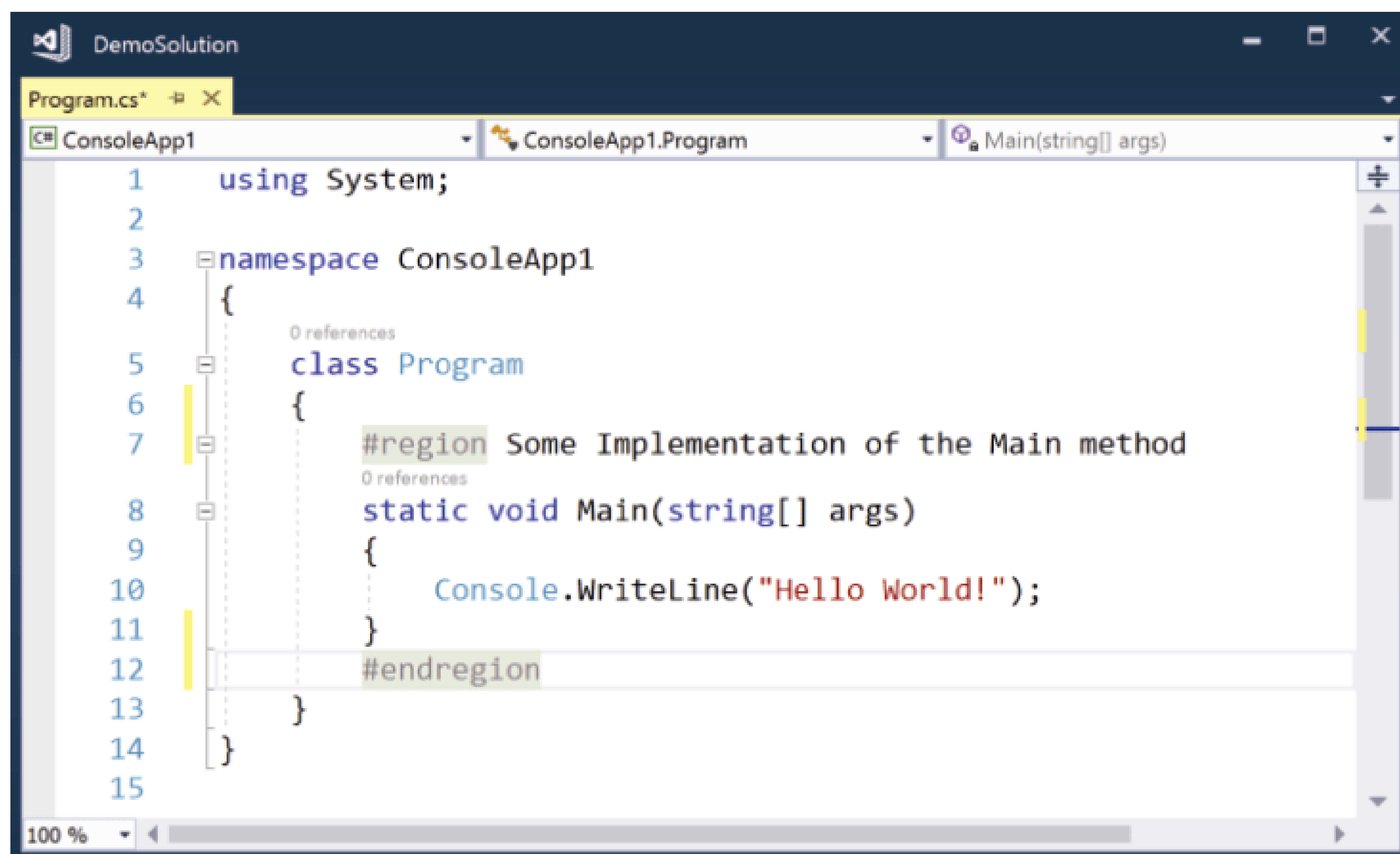



图 18-25

代码编辑器自动检测#region 块，并通过#region 指令放置一个新的减号标识，这允许关闭该区域。封闭区域中的这段代码允许编辑器关闭它(如图 18-26 所示)，在#region 指令中用指定的注释标记这个区域。然而，编译器会忽略这些指令，跟往常一样编译 Main()方法。

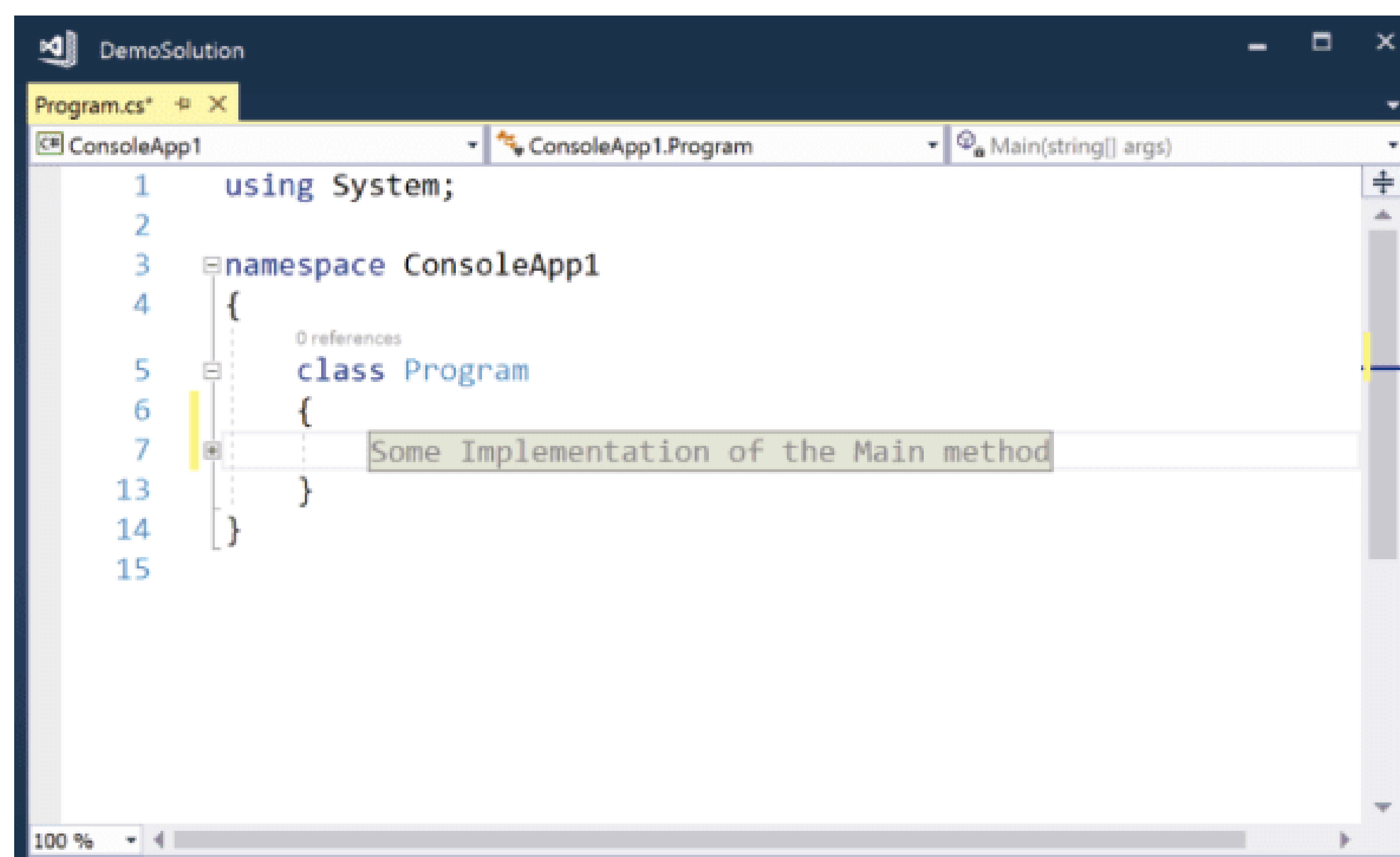



图 18-26

### 2. 在编辑器中导航

编辑器的顶行是三个组合框。右边的组合框允许导航输入的类型成员。中间的组合框允许导航类型。左边的组合框允许在不同的应用程序或框架之间导航。例如，如果正在处理一个共享项目的源代码，在编辑器的左边组合框中，可以选择使用共享项目的一个项目，查看所选项目的活跃代码。不为所选项目编译的代码会暗显。使用 C#预处理器命令可以为不同的平台创建代码段。

### 3. IntelliSense

除了可折叠编辑器的功能之外，Visual Studio 的代码编辑器也集成了 Microsoft 流行的 IntelliSense 功能。它不仅减少了输入量，还确保使用正确的参数。IntelliSense 会记住首选项，并从这些选项开始提供列表，而不是使用 IntelliSense 提供的有时相当长的列表。

代码编辑器甚至在编译代码之前就对代码进行语法检查，用短波浪线指示错误。将鼠标指针悬停在带有下划线的文本上，会弹出一个包含了错误描述的小方框。

### 4. CodeLens

用户可能修改了一个方法，但忘了调用它的方法。现在很容易找到调用者。引用数会直接显示在编辑器中，如图 18-27 所示。单击引用链接时，会打开 CodeLens，以便查看调用者的代码，并导航到它们。

如果使用 Git 或 TFS 把源代码签入到源代码控制系统中，例如 Visual Studio Online，也可以看到作者和所进行的更改。如果正在使用单元测试(在第 28 章中介绍)，就可以看到成功和失败的测试运行数量，可以立即切换到详细的信息。

> **注意：**
> CodeLens 不能用于 Visual Studio Community 版本。

### 5. 使用代码片段

代码片段提升了代码编辑器的工作效率，仅需要在编辑器中写入 cw<tab><tab>，编辑器就会创建 Console.WriteLine();。Visual Studio 自带很多代码片段：

- 使用快捷方式 do、for、forr、foreach 和 while 创建循环
- 使用 equals 实现 Equals 方法
- 使用 attribute 和 exception 来创建 Attribute 和 Exception 派生类型等

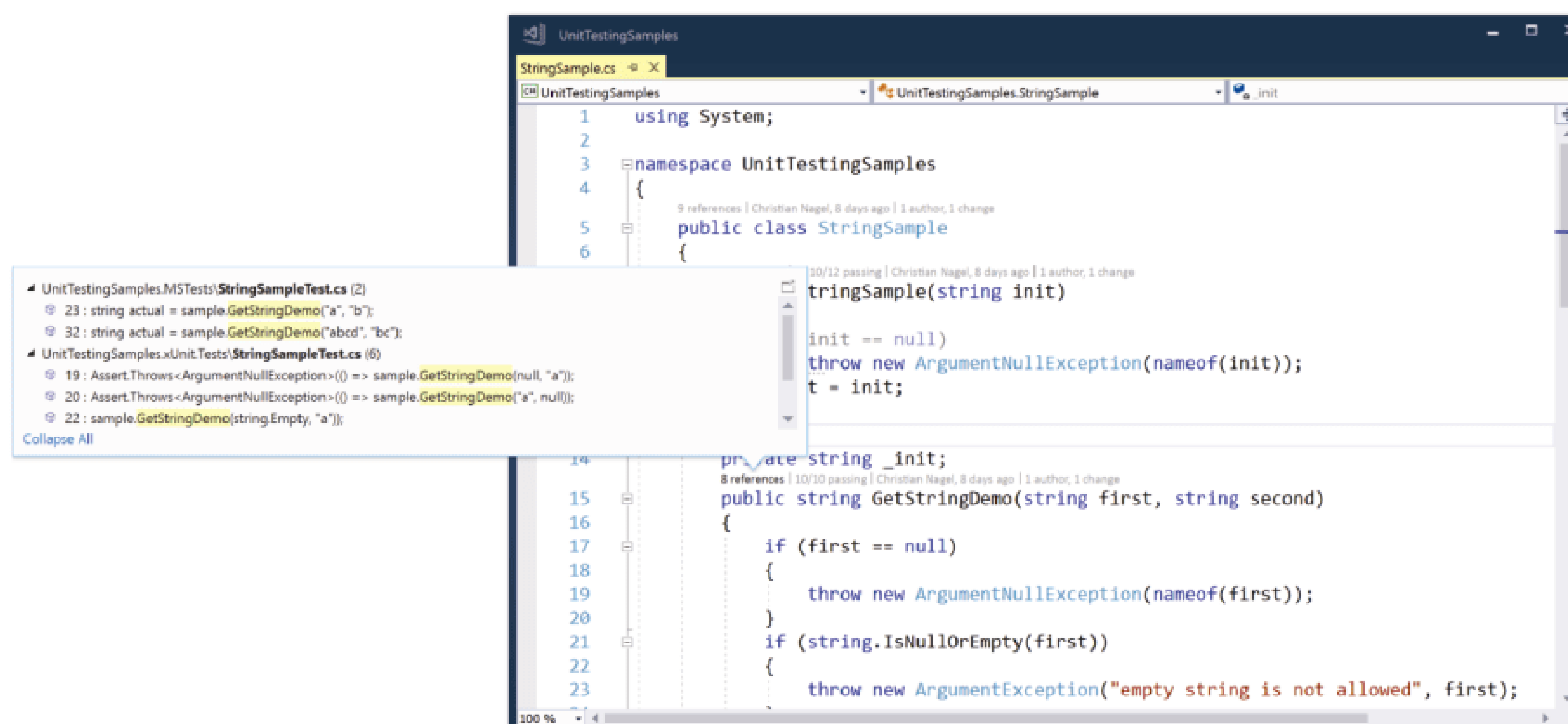


图 18-27

选择 Tools | Code Snippets Manager，在打开的 Code Snippets Manager 中可以看到所有可用的代码片段(如图 18-28 所示)。也可以创建自定义的代码片段。

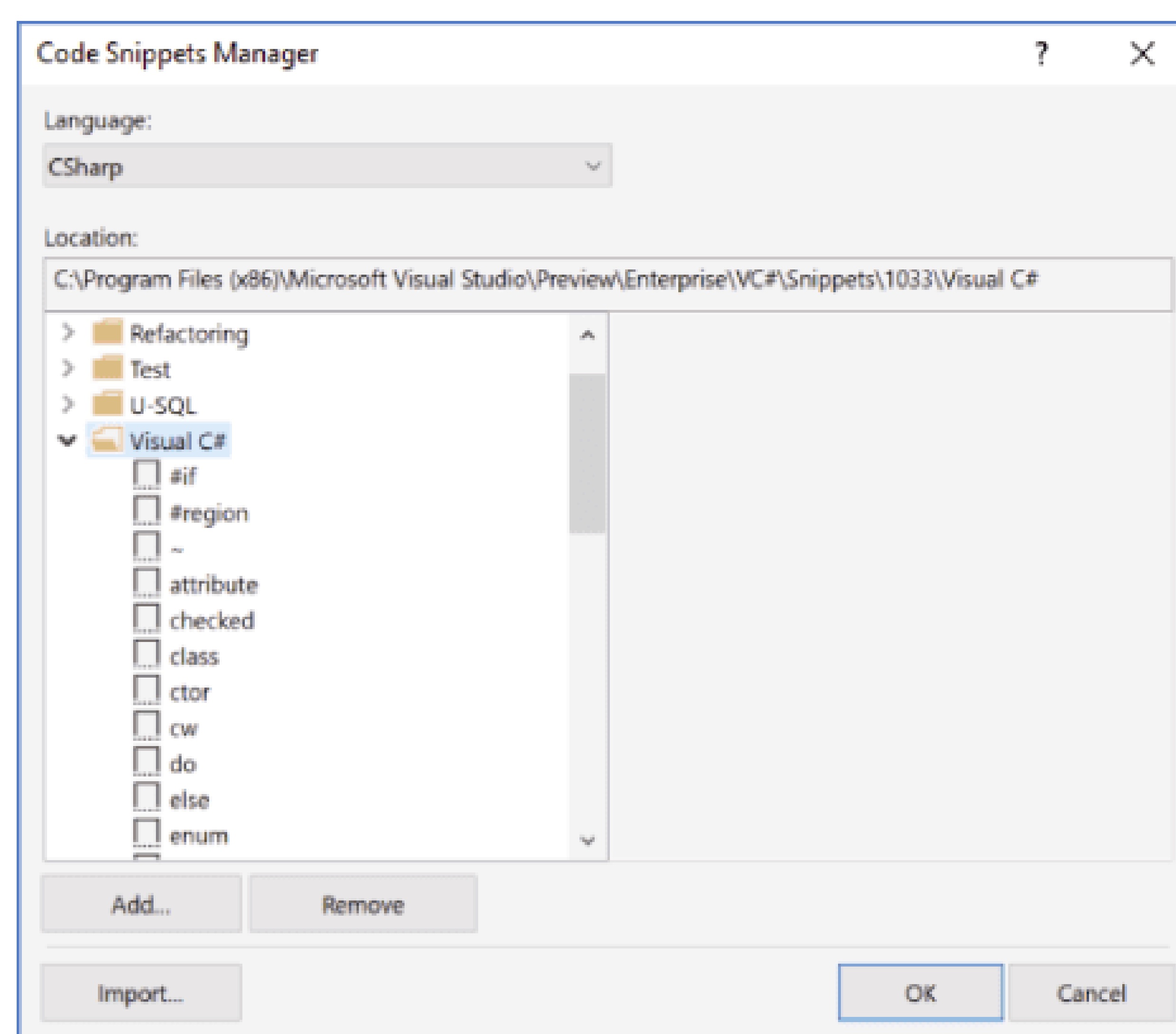


图 18-28

### 6. EditorConfig

Visual Studio 支持不同的编码风格。在 Visual Studio 的早期版本中，可以配置环境的编码风格。但是，处理不同的项目时，可能需要不同的编码风格。如果在 Tools | Options | Text Editor | General 中配置 Follow Project Coding Conventions 选项，EditorConfig 就支持不同的编码风格。

要使用 EditorConfig，可以将.editorconfig 文件添加到项目目录或子目录中。该配置适用于此目录和子目录中的所有源代码文件。在子目录中，可以添加更多的.editorconfig 文件，这些文件可以覆盖父目录的配置。

ASP.NET Core MVC 团队使用的.editorconfig 文件如下代码片段所示。在这个文件中，可以根据不同的文件扩展名，定义缩进的大小，还可以定义编码指南——例如，var 是应该首选还是无效；应该或不应该使用 this 限定符，C#定义的类型是否应该优于.NET 类型，是否应该允许 throw 表达式等。通过这些设置，可以判断编辑器应该生成建议、警告还是错误：

```
# EditorConfig is awesome:http://EditorConfig.org

# top-most EditorConfig file
root = true

# Don't use tabs for indentation.
[*]
```

```
indent_style = space
# (Please don't specify an indent_size here; that has too many unintended
# consequences.)

# Code files
[*.{cs,csx,vb,vbx}]
indent_size = 4

# Xml project files
[*.{csproj,vbproj,vcxproj,vcxproj.filters,proj,projitems,shproj}]
indent_size = 2

# Xml config files
[*.{props,targets,ruleset,config,nuspec,resx,vsixmanifest,vsct}]
indent_size = 2

# JSON files
[*.json]
indent_size = 2

# Dotnet code style settings:
[*.cs]
# Sort using and Import directives with System.* appearing first
dotnet_sort_system_directives_first = true

# Don't use this. qualifier
dotnet_style_qualification_for_field = false:suggestion
dotnet_style_qualification_for_property = false:suggestion

# use int x = .. over Int32
dotnet_style_predefined_type_for_locals_parameters_members = true:suggestion

# use int.MaxValue over Int32.MaxValue
dotnet_style_predefined_type_for_member_access = true:suggestion

# Require var all the time.
csharp_style_var_for_built_in_types = true:suggestion
csharp_style_var_when_type_is_apparent = true:suggestion
csharp_style_var_elsewhere = true:suggestion

# Disallow throw expressions.
csharp_style_throw_expression = false:suggestion

# Newline settings
csharp_new_line_before_open_brace = all
csharp_new_line_before_else = true
csharp_new_line_before_catch = true
csharp_new_line_before_finally = true
csharp_new_line_before_members_in_object_initializers = true
csharp_new_line_before_members_in_anonymous_types = true
```

使用这个.editorconfig 文件，输入 int x = 42，就可以看到一个建议，如图 18-29 所示。请注意，因为可能需要在更改.editorconfig 后重新打开源代码，以激活新的配置。

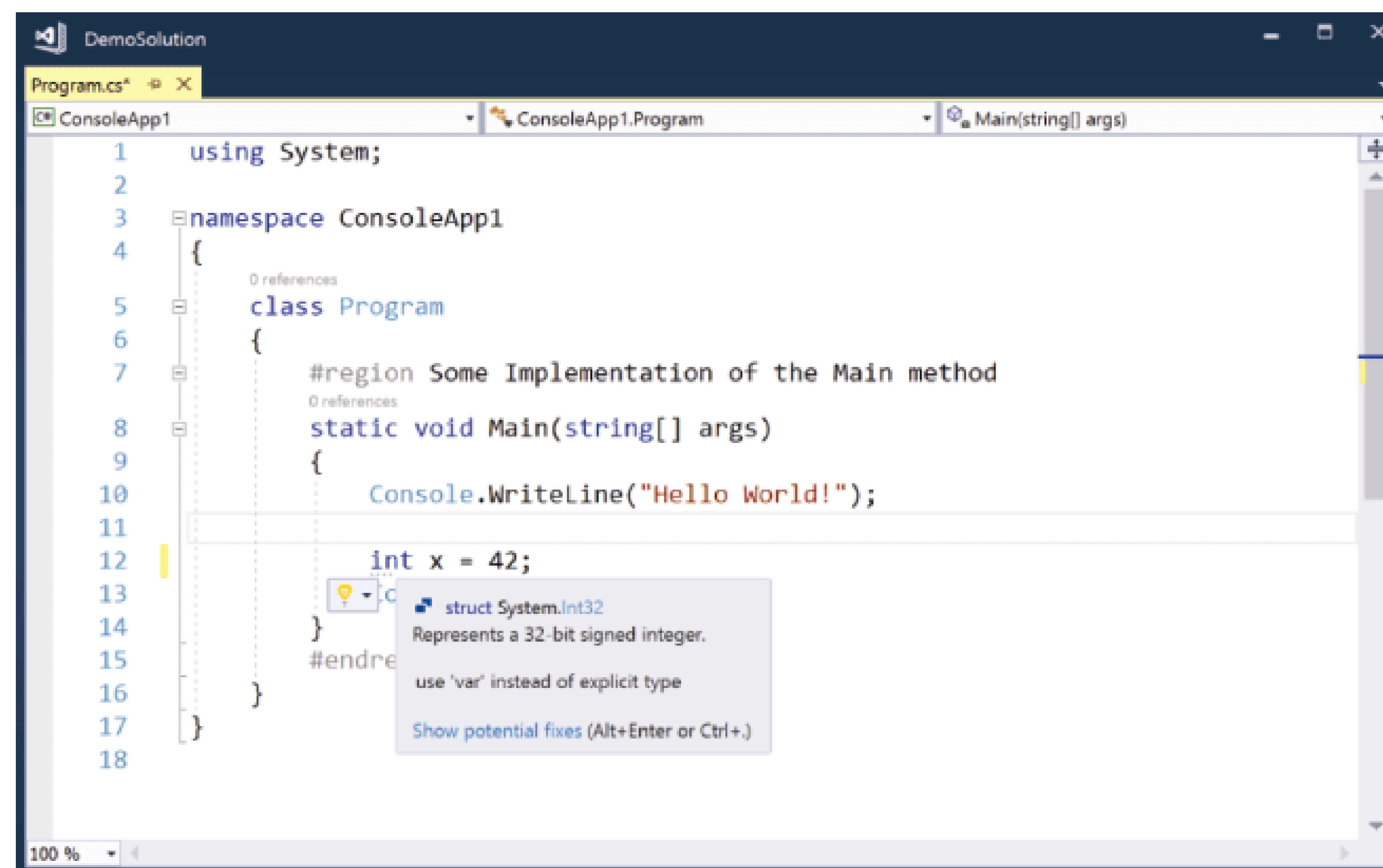



图 18-29

> **注意：**
> 在 http://editorconfig.org 上可以获得关于文件格式、各种编辑器的插件、editorconfig 项目页面的更多信息。

### 18.3.3　学习和理解其他窗口

除了代码编辑器和 Solution Explorer 外，Visual Studio 还提供了许多其他窗口，允许从不同的角度来查看或管理项目。

> **注意：**
> 本节的其余部分介绍其他几个窗口。如果这些窗口在屏幕上不可见，可以在 View 菜单中选择它们。要显示设计视图或代码编辑器，可以右击 Solution Explorer 中的文件名，并选择上下文菜单中的 View Designer 或 View Code；也可以选择 Solution Explorer 顶部工具栏中的对应项。设计视图和代码编辑器共用同一个选项卡式窗口。

#### 1. 使用设计视图窗口

如果设计一个用户界面应用程序，如 Windows 应用程序或类库(Universal Windows)，则可以使用设计视图窗口。这个窗口显示窗体的可视化概览。设计视图窗口经常和工具箱窗口一起使用。工具箱包含许多.NET 组件，可以将它们拖放到程序中。工具箱的组件会根据项目类型而有所不同。图 18-30 显示了 Windows 应用程序中的数据项。

要将自定义的类别添加到工具箱，请执行如下步骤：

(1) 右击任何一个类别。

(2) 选择上下文菜单中的 Add Tab。

可以将代码片段移动到工具箱中的项中，这样就可以方便地访问它们。也可以选择上下文菜单中的 Choose Items，在工具箱中放置其他工具，这尤其适合于添加自定义的组件或工具箱默认没有显示的通用窗口组件，如图 18-31 所示。

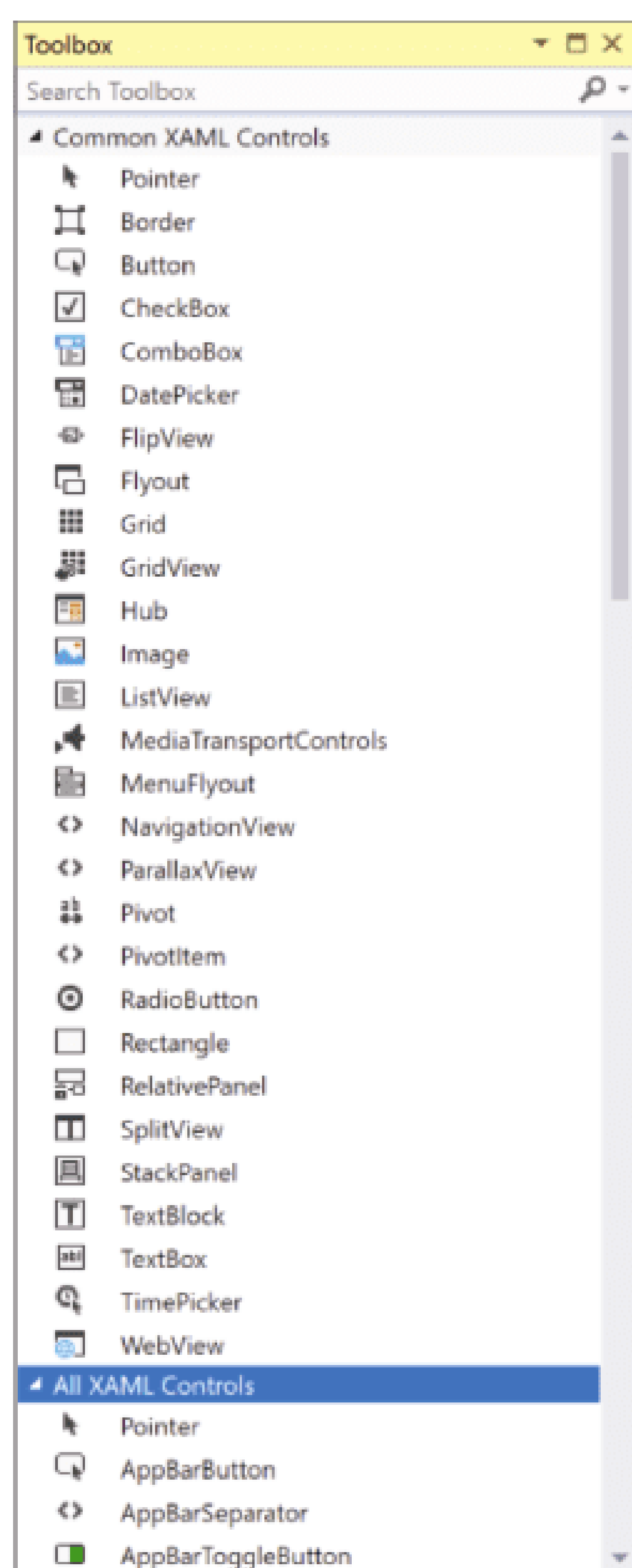



图 18-30

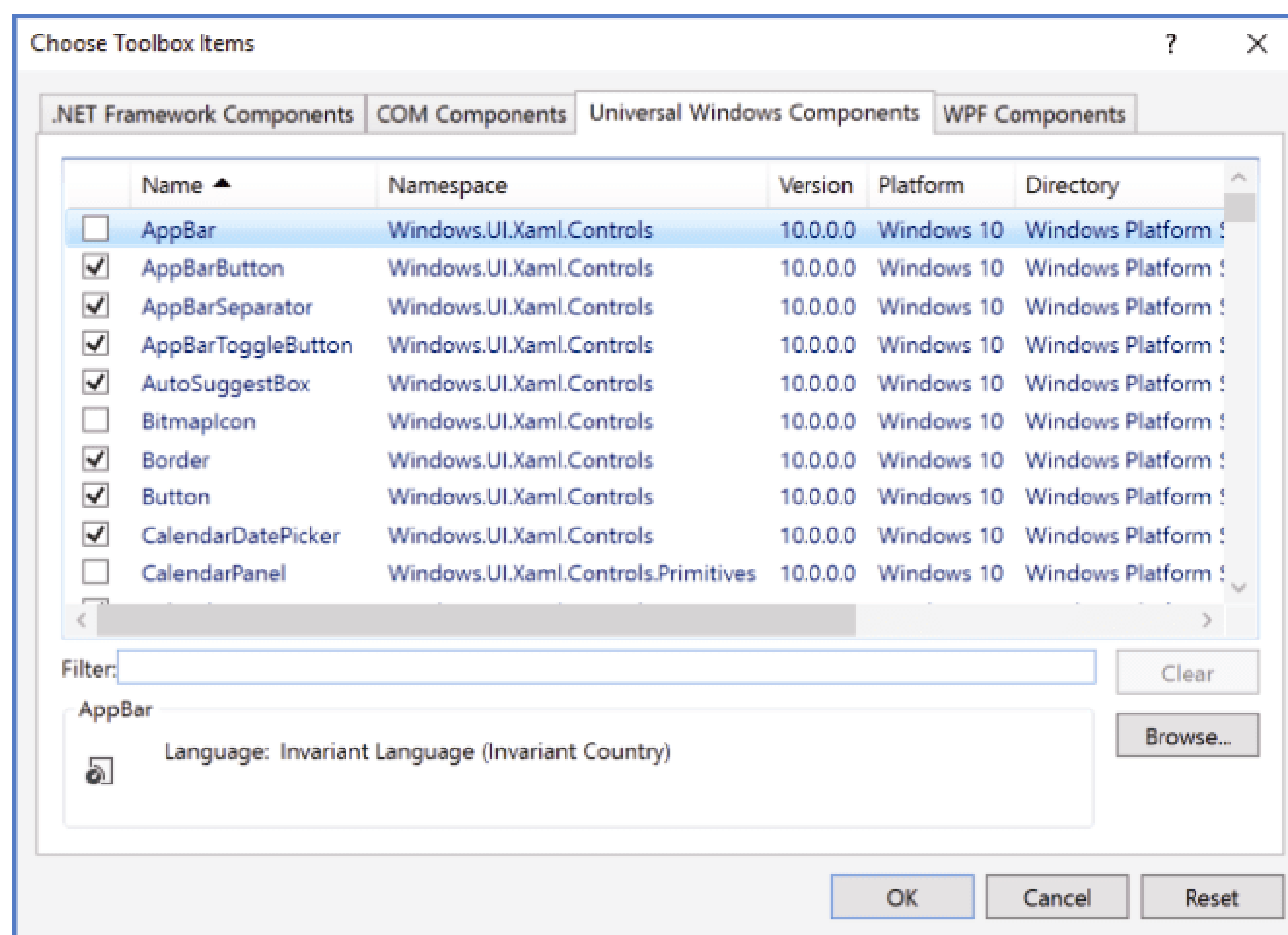



图 18-31

### 2. 使用 Properties 窗口

如本书第 I 部分所述，.NET 类可以实现属性。Properties 窗口可用于项目、文件和使用设计视图选择的项。图 18-32 显示了 Windows 应用程序中一个控件的 Properties 视图。

在这个窗口中可以看到一项的所有属性，并对其进行相应的配置。一些属性可以通过在文本框中输入文本来改变，一些属性有预定义的选项，一些属性有自定义的编辑器。也可以在 Properties 窗口中添加事件处理程序。

### 3. 使用类视图窗口

Solution Explorer 可以显示类和类的成员，这是类视图的一般功能(如图 18-33 所示)。要调用类视图，可选择 View | Class View。类视图显示代码中的名称空间和类的层次结构。它提供了一个树型结构，可以展开该结构来查看名称空间下包含哪些类，类中包含哪些成员。

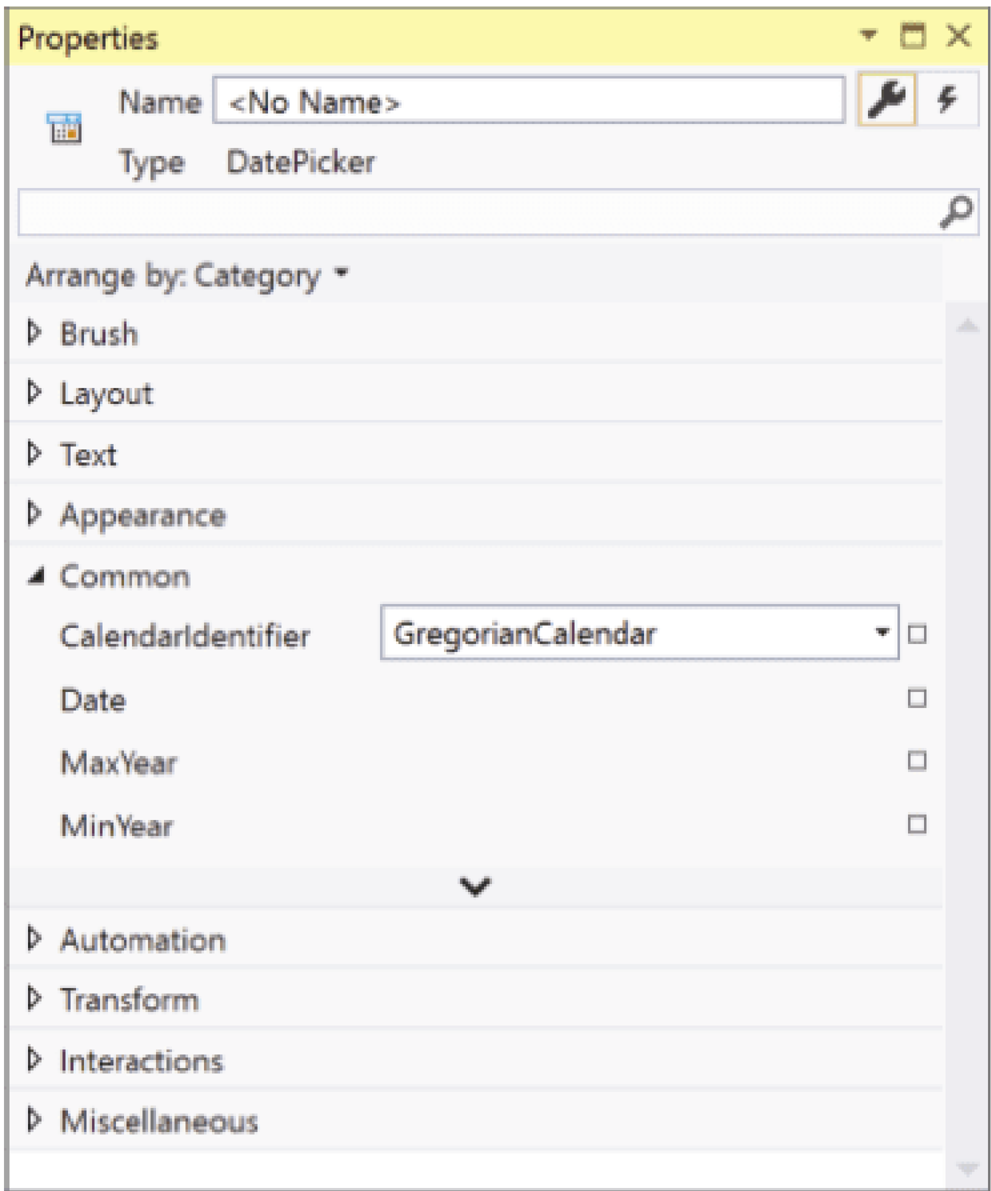



图 18-32

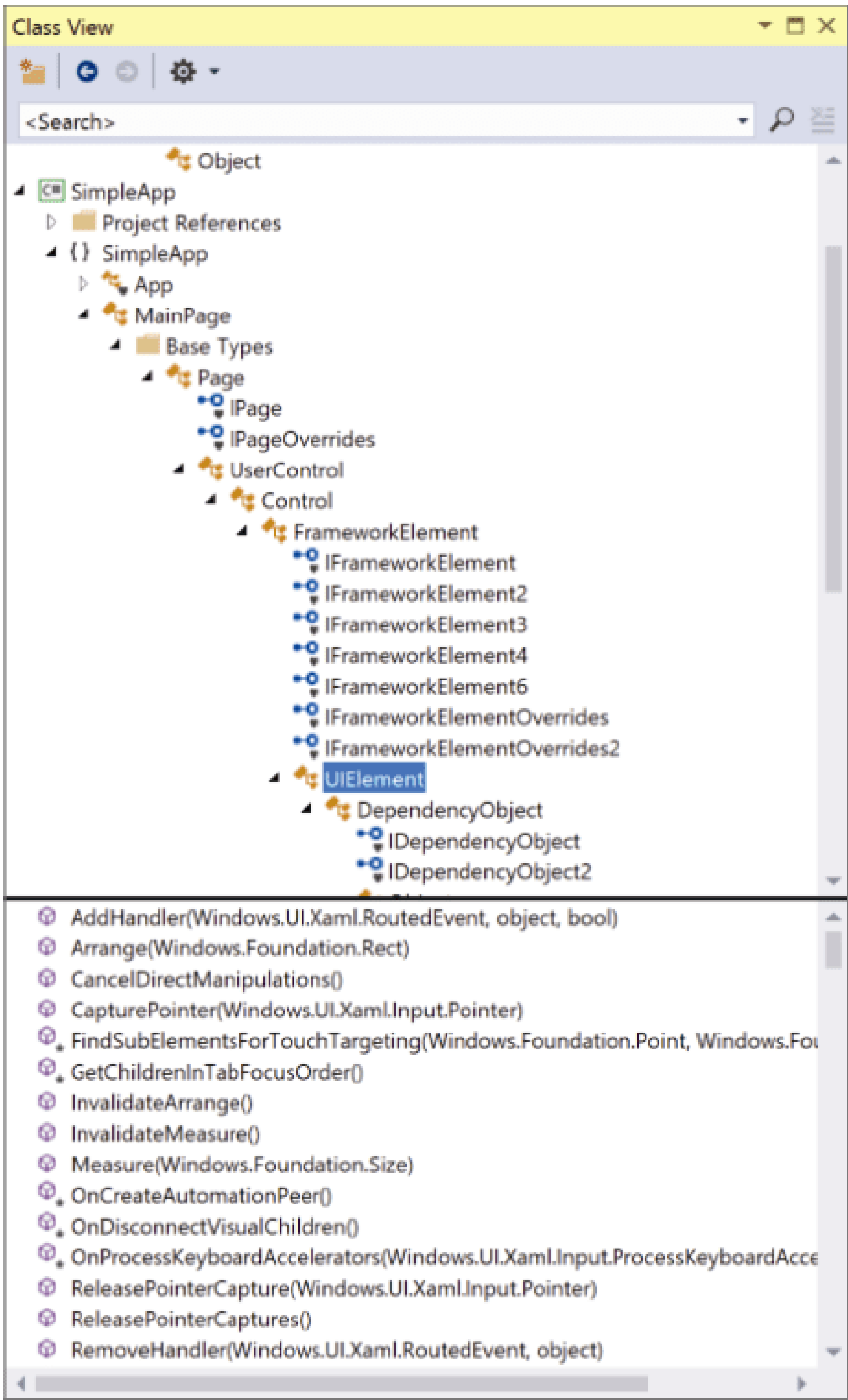



图 18-33

类视图的一个杰出功能是，如果右击任何有权访问其源代码的项的名称，然后选择上下文菜单中的 Go To Definition 命令，就会转到代码编辑器中的项定义。另外，在类视图中双击该项(或在代码编辑器中右击想要的项，并从上下文菜单中选择相同的选项)，也可以查看该项的定义。上下文菜单还允许给类添加字段、方法、属性或索引器。换句话说，在对话框中指定相关成员的详细信息，就会自动添加代码。这个功能对于添加属性和索引器非常有用，因为它可以减少相当多的输入量。

### 4. 使用 Object Browser 窗口

在.NET 环境中编程的一个重要方面是能够找出基类或从程序集引用的其他库中有哪些可用的方法和其他代码项。这个功能可通过 Object Browser 窗口获得(参见图 18-34)。在 Visual Studio 2017 中选择 View 菜单中的

Object Browser，可以访问这个窗口。使用这个工具，可以浏览并选择现有的组件集，如.NET Framework 4.0 到 4.7.1 版本、适用于 Windows 运行库的.NET Portable Subsets 以及.NET for UWP，并查看这个子集中可用的类和类成员。在 Browse 下拉框中选择 Universal Windows，来选择 Windows 运行库，也可以找到这个用于 UWP 应用程序的原生新 API 的所有名称空间、类型和方法。

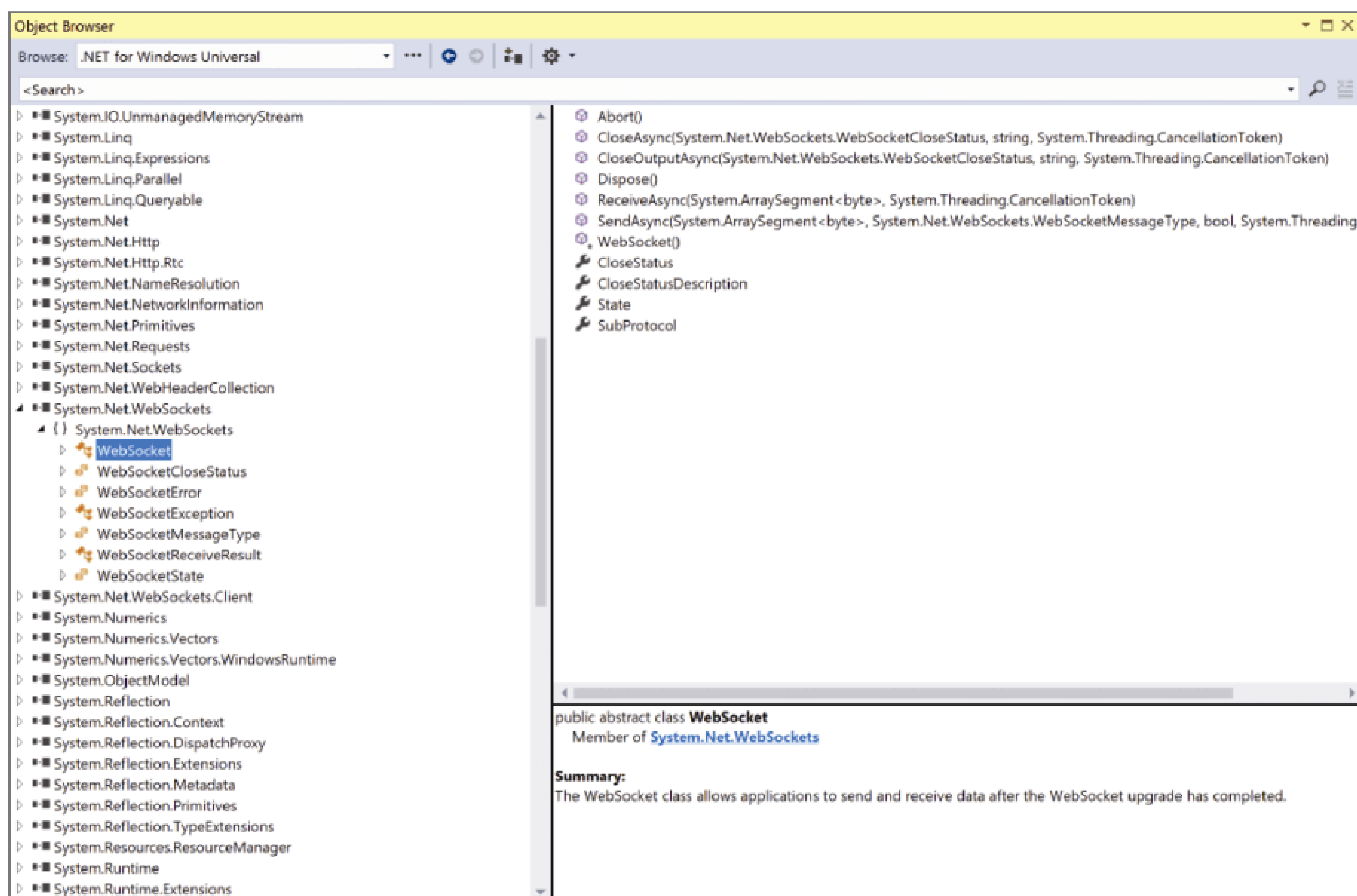


图 18-34

**注意：**

不要因为下拉列表中提供的许多选择没有显示任何结果而对 Object Browser 感到厌烦，这是根据设计而定的。

### 5. 使用 Server Explorer 窗口

使用 Server Explorer 窗口，如图 18-35 所示，可以在编码时找出计算机在网络中的相关信息。在该窗口的 Servers 部分中，可以找到服务运行情况的信息(这对于开发 Windows 服务是非常有用的)，创建新的性能计数，访问事件日志。在 Data Connections 部分中不仅能够连接现有数据库，查询数据，还可以创建新的数据库。Visual Studio 2017 也有一些内置于 Server Explorer 的 Windows Azure 信息，包括 App Services、Virtual Machines、Notifications、Storage 等选项。

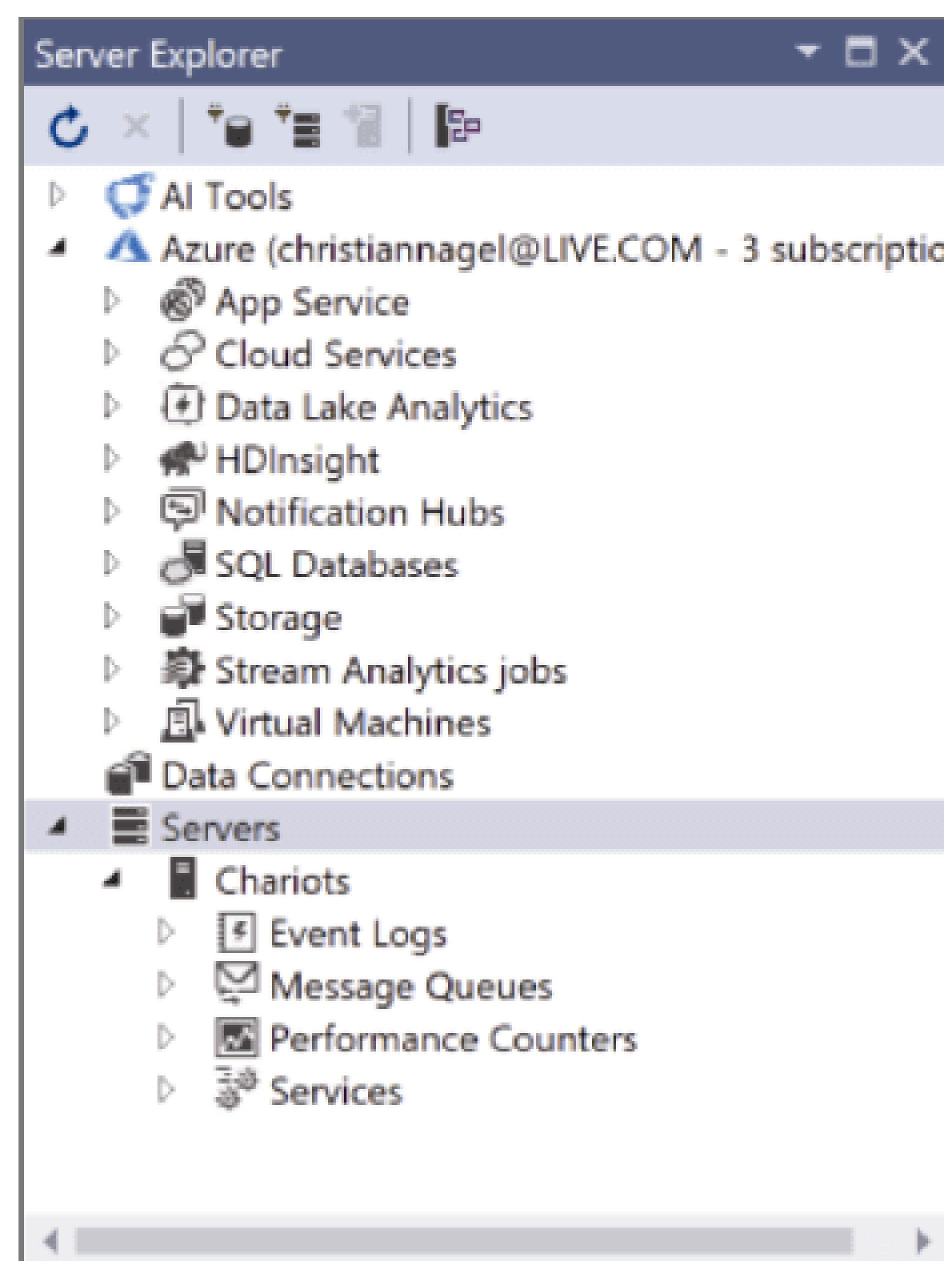


图 18-35

### 6. 使用 Cloud Explorer

如果安装了 Azure SDK，那么 Cloud Explorer (见图 18-36)是一个可用于 Visual Studio 2017 的新浏览器。使用 Cloud Explorer 可以访问 Microsoft Azure 订阅，访问资源，查看日志文件，连接调试器，启动和停止服务，直接进入 Azure 门户，启动调试会话。

#### 7. 使用 Document Outline 窗口

可用于 WPF 和 UWP 应用程序的一个窗口是 Document Outline。如图 18-37 所示，在这个窗口中打开了网上附加第 1 章的一个应用程序，从中可以查看 XAML 元素的逻辑结构和层次结构，锁定元素以防止其无意中被修改，在层次结构中轻松地移动元素，在新的元素容器中分组元素和改变布局类型。

使用这个工具还可以创建 XAML 模板，图形化地编辑数据绑定。

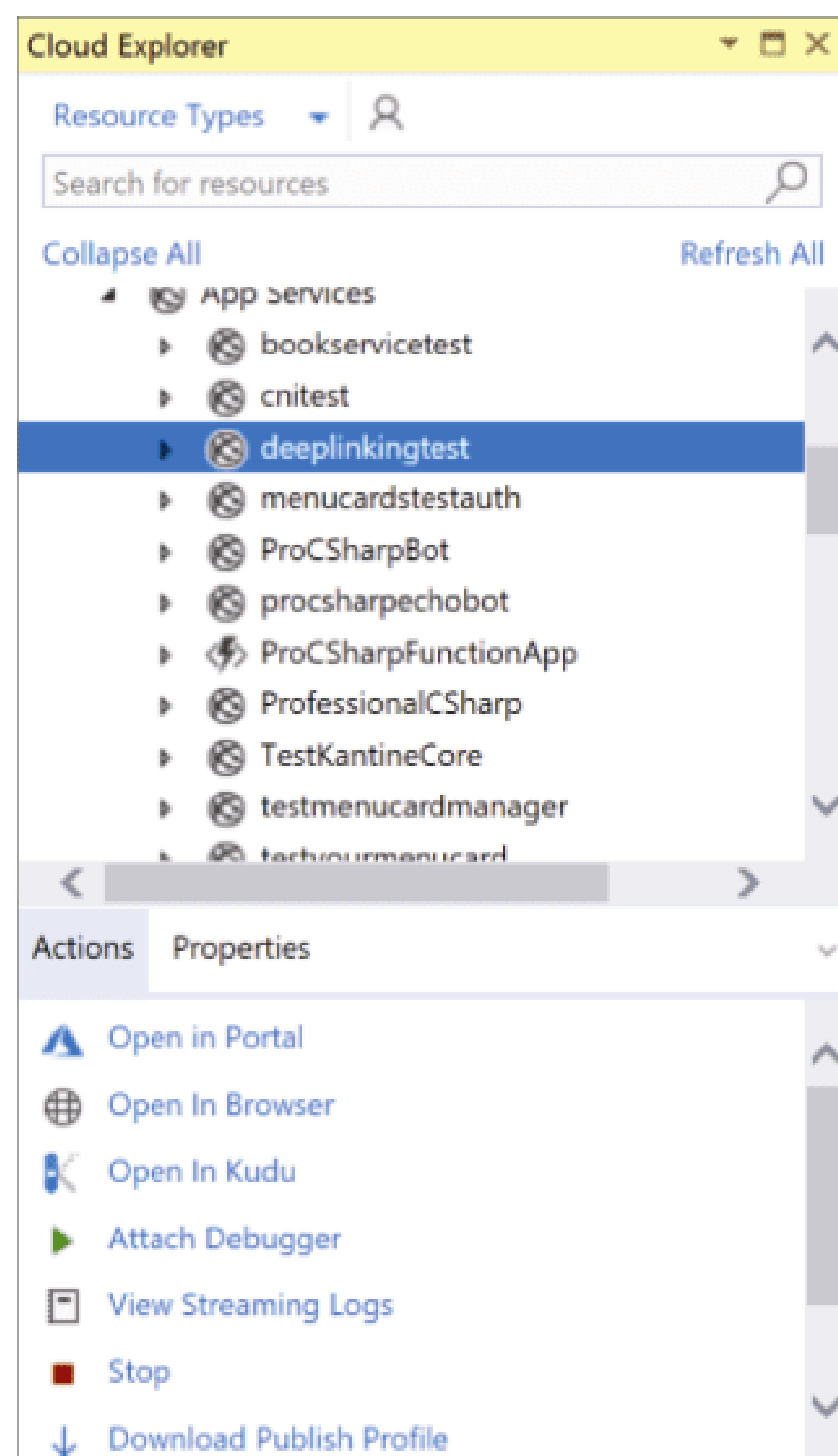



图 18-36

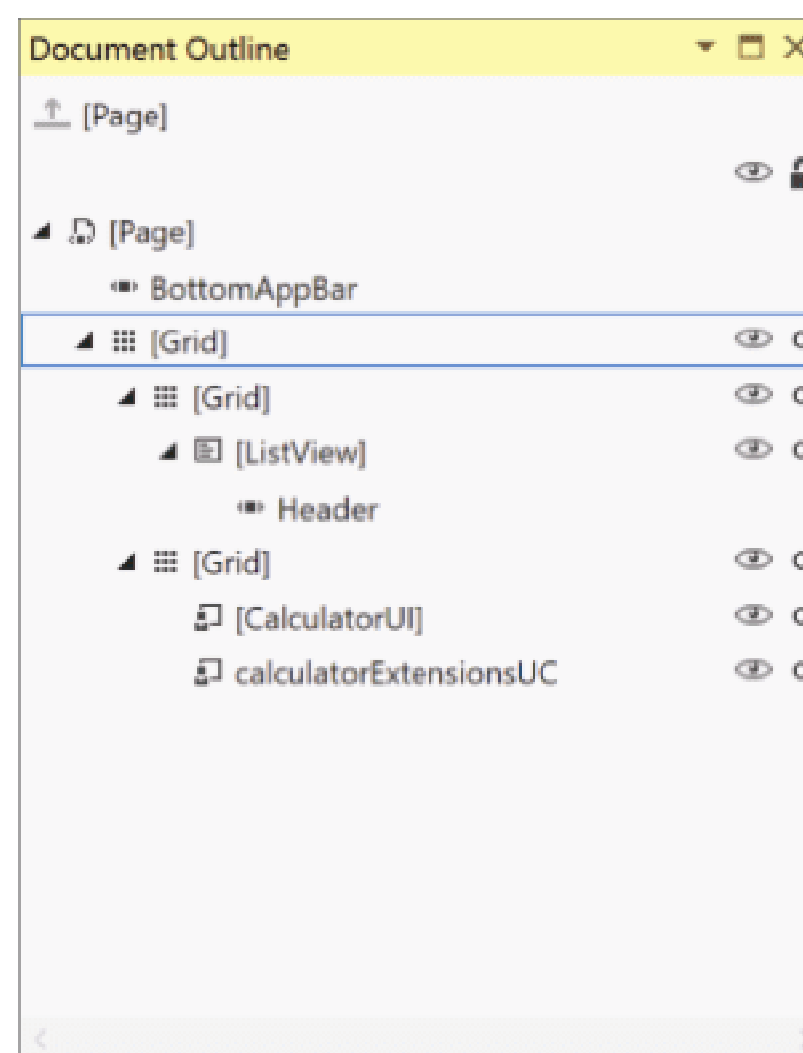



图 18-37

### 18.3.4　排列窗口

学习 Visual Studio 时会发现，许多窗口有一个有趣的功能会让人联想到工具栏。尤其是，它们都可以浮动(也可以显示在第二个显示器上)，也可以停靠。当它们停靠时，在每个窗口右上角的最小化按钮旁边会显示一个类似图钉的额外图标。这个图标的作用确实像图钉，它可以用来固定打开的窗口。固定窗口(图钉是垂直显示的)的行为与平时使用的窗口一样。但当它们取消固定时(图钉是水平显示的)，则窗口只有获得焦点才会打开。当失去焦点时(因为单击或者移动鼠标到其他地方)，它们会快速退出到 Visual Studio 应用程序的主边框内。固定窗口和取消固定窗口提供了另一种方式来更好地利用屏幕上有限的空间。

Visual Studio 2017 中的一个特性是，可以存储不同的布局。用户很有可能运行在不同的环境中。例如，在办公室笔记本电脑可能连接到两个大屏幕上，但在飞机上编程时，就只有一个屏幕。过去，可能总是根据需要安排窗口，必须一天几次地改变窗口的布局。可能需要不同布局的另一个场景是做网络开发，创建 UWP 和 Xamarin 应用程序。现在可以保存布局，轻松地从一个布局切换到另一个。在 Window 菜单中选择 Save Window Layout，保存当前的工具布局。使用 Window | Apply Window Layout，选择一个保存的布局，把窗口安排为保存它们时的布局。

## 18.4　构建项目

Visual Studio 不仅可以编写项目，它实际上是一个 IDE，管理着项目的整个生命周期，包括生成或编译解决方案。本节讨论如何用 Visual Studio 生成项目。

### 18.4.1 构建、编译和生成代码

讨论各种构建选项之前，先要弄清楚一些术语。从源代码转换为可执行代码的过程中，经常看到 3 个不同的术语：构建、编译和生成。这 3 个术语的起源反映了一个事实：直到最近，从源代码到可执行代码的过程涉及多个步骤(在 C++中仍然如此)。这主要是因为一个程序包含了大量的源文件。

例如，在 C++中，每个源文件都需要单独编译。这就产生了所谓的对象文件，每个对象文件包含类似于可执行代码的内容，但每个对象文件只与一个源文件相关。要生成一个可执行文件，这些对象文件需要连接在一起，这个过程官方称为链接。这个合并过程通常称为构建代码(至少在 Windows 平台上是如此)。然而，在 C#术语中，编译器比较复杂，能够将所有的源文件当作一个块来读取和处理。因此，没有真正独立的链接阶段，所以在 C#上下文中，术语“构建”和“编译”可以互换使用。

术语“生成”的含义与“构建”基本相同，虽然它在 C#上下文中没有真正使用。术语“生成”起源于旧的大型机系统，在该系统中，当一个项目由许多源文件组成时，就在一个单独的文件中写入指令，告诉编译器如何构建项目：包含哪些文件和链接什么库等。这个文件通常称为生成文件，在 UNIX 系统上它仍然是非常标准的文件。事实上，MSBuild 项目文件和旧的生成文件非常类似，它只是一个新的高级 XML 变体。在 MSBuild 项目中，可以使用 MSBuild 命令，将项目文件当作输入，来编译所有的源文件。使用构建文件非常适合于在一个单独的构建服务器上进行构建，其中所有的开发人员仅需要签入他们的代码，构建过程会在深夜自动完成。第 1 章介绍了.NET Core 命令行 (CLI)工具，该命令行建立了.NET Core 环境，现在在后台中使用 MSBuild。

### 18.4.2 调试版本和发布版本

C++开发人员非常熟悉生成两个版本的这种思想，有 Visual Basic 开发背景的开发人员也不会十分陌生。其关键在于：可执行文件在调试时的目标和行为应与正式发布时不同。准备发布软件时，可执行文件应尽可能小而快。但是，这两个目标与调试代码时的需求不兼容，在接下来的小节中将看到这一点。

#### 1. 优化

在高性能方面，编译器对代码进行的多次优化起到了一定的作用。这意味着编译器在编译代码时，会在代码实现细节中积极找出可以修改的地方。编译器所做的修改并不会改变整体效果，但是会使程序更加高效。例如，假设编译器遇到了下面的源代码：

```
double InchesToCm(double ins) => ins * 2.54;
// later on in the code
Y = InchesToCm(X);
```

就可能把它们替换为下面的代码：

```
Y = X * 2.54;
```

类似地，编译器可能把下面的代码：

```
{
  string message = "Hi";
  Console.WriteLine(message);
}
```

替换为：

```
Console.WriteLine("Hi");
```

这样，编译器就不需要在此过程中声明任何非必要的对象引用。

C#编译器会进行怎样的优化无从判断，我们也不知道前两个例子中的优化在特定情况中是否会实际发生，因为编译器的文档没有提供这类细节。不过，对于 C#这样的托管语言，上述优化很可能在 JIT 编译时发生，而不是在 C#编译器把源代码编译为程序集时发生。显然，由于专利原因，编写编译器的公司通常不愿意过多地说明他们使用了什么技巧。注意，优化不会影响源代码，而只影响可执行代码的内容。通过前面的示例，可以基本了解优化产生的效果。

问题在于，虽然示例代码中的优化可以加快代码的运行速度，但是它们也增加了调试的难度。在第一个例子中，假设想要在 InchesToCm()方法中设置一个断点，了解该方法的工作机制。如果在编译器做了优化后，可执行代码中不再包含 InchesToCm()方法，怎么可能进行这种操作呢？同样，如果编译后的代码中不再包含 Message 变量，又如何监视该变量的值？

#### 2. 调试器符号

在调试过程中，经常需要查看变量的值，这时使用的是它们在源代码中的名称。问题是可执行代码一般不包含这些名称——编译器用内存地址代替了变量名称。.NET 在一定程度上改变了这种情况，使程序集中的某些项以名称的形式存储，但是这只适用于少量的项(例如公有类和方法)，而且这些名称在 JIT 编译程序集后也仍然会被移除。如果让调试器显示变量 HeightInInches 的值，但是编译器在查看可执行代码时只看到了地址，而没有看到任何对名称 HeightInInches 的引用，自然就得不到期望的结果。

因此，为了正确地调试，需要在可执行文件中提供一些额外的调试信息。这些信息包含变量名和代码行信息，允许调试器确定可执行机器汇编语言指令与源代码中的哪些指令对应。但是，不应该在发布版本中包含这些信息，这既是出于专利考虑(提供调试信息会方便其他人反汇编代码)，也是因为包含调试信息会增加可执行文件的大小。

#### 3. 其他源代码调试指令

一个相关问题是，在调试时，程序经常包含一些额外的代码行，用于显示关键的调试信息。显然，在发布软件前，需要从可执行代码中彻底删除这些相关指令。手动删除是可以的，但是如果能以某种方式标记这些语句，让编译器在编译发布代码时自动忽略它们，不是更方便吗？本书的第 I 部分提到，在 C#中，定义合适的预处理器指令，再结合使用 Conditional 特性(所谓的条件编译)，就可以实现这种操作。

所有这些因素综合到一起，决定了几乎所有商业软件的编译调试方式与最终交付产品的编译方式是稍有区别的。Visual Studio 能够处理这种区别，因为 Visual Studio 在编译代码时，会存储应传递给编译器的所有编译选项信息。为了支持不同类型的构建版本，Visual Studio 需要存储多组编译选项。这些不同的版本信息集合称为配置。在创建项目时，Visual Studio 会自动提供两种配置：调试和发布。

- **调试**：这种配置通常指定编译器不优化编译过程，可执行文件应该包含额外的调试信息，编译器假定调试预处理器指令 Debug 是存在的，除非源代码中显式使用了#undefined Debug 指令。
- **发布**：这种配置指定编译器应优化编译过程，可执行文件不应包含额外的调试信息，编译器不应假定源代码包含特定的预处理器符号。

还可以定义自己的配置，例如设置软件的专业级版本和企业级版本。过去，由于 Windows NT 支持 Unicode 字符编码，但 Windows 95 不支持，因此 C++项目经常使用 Unicode 配置和 MBCS(Multi-Byte Character Set，多字节字符集)配置。

### 18.4.3　选择配置

Visual Studio 存储了多个配置的细节，那么在准备生成一个项目时，如何决定使用哪个配置？答案是，项目总是有一个活动的配置，当要求 Visual Studio 生成项目时，就使用这个配置。注意，活动配置是针对每个项目、而不是每个解决方案设置的。

在创建项目时，默认情况下 Debug 配置是活动配置。如果想修改活动配置，可以单击 Build 菜单，选择 Configuration Manager 菜单项。在 Visual Studio 主工具栏的下拉菜单中也可以找到此选项。

### 18.4.4　编辑配置

除了选择活动配置外，还可以查看及编辑配置。为此，在 Solution Explorer 中选择相关的项目，然后选择 Project 菜单中的 Properties 菜单项，这会打开一个复杂的对话框。打开该对话框的另一个方法是，在 Solution

Explorer 中右击项目名称，然后从上下文菜单中选择 Properties。

这个对话框包含多个选项卡，用于选择要查看或编辑的常规属性类别。由于篇幅原因，本节不展示所有的属性类别，只介绍其中两个最重要的选项卡。

根据项目类型的不同，可用的选项完全不同。首先，图 18-38 显示了 UWP 应用程序的属性，其中显示了可用属性的选项卡式视图。这个屏幕截图显示了常规应用程序设置。

需要注意的一点是，可以选择程序集的名称以及使用新项生成的默认名称空间。单击 Assembly Information 按钮，可以输入版本号、标题、描述、公司和其他信息。单击 Package Manifest 按钮将切换到第 33 章中介绍的 Package Manifest 编辑器。

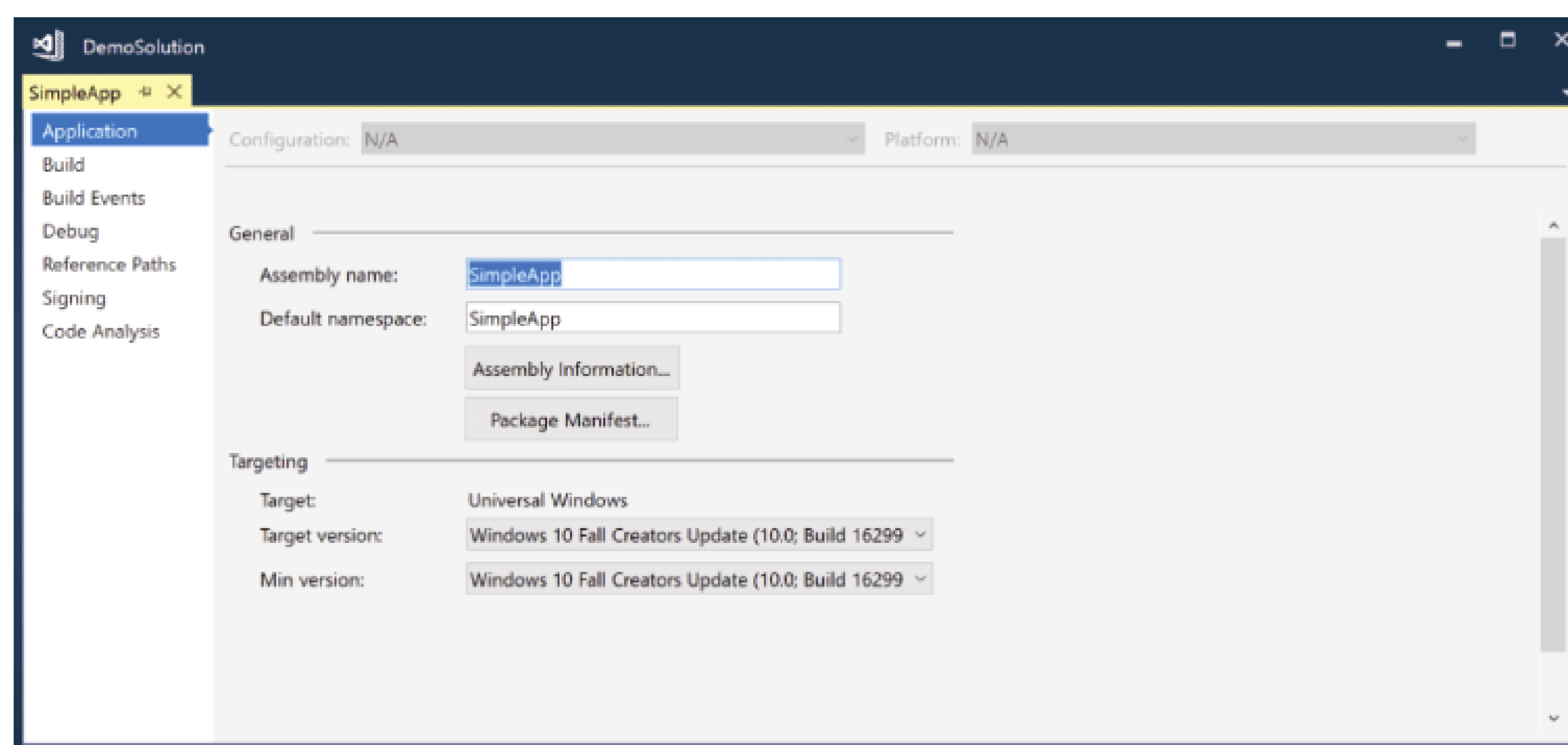



图 18-38

图 18-39 显示了.NET Core Console 应用程序的配置。还可以看到 Application 设置，但是这个屏幕看起来不同。程序集名称和默认名称空间也是可配置的，但是这里可以选择目标框架的版本、应用程序的输出类型和启动对象(如果有多个 Main()方法)。

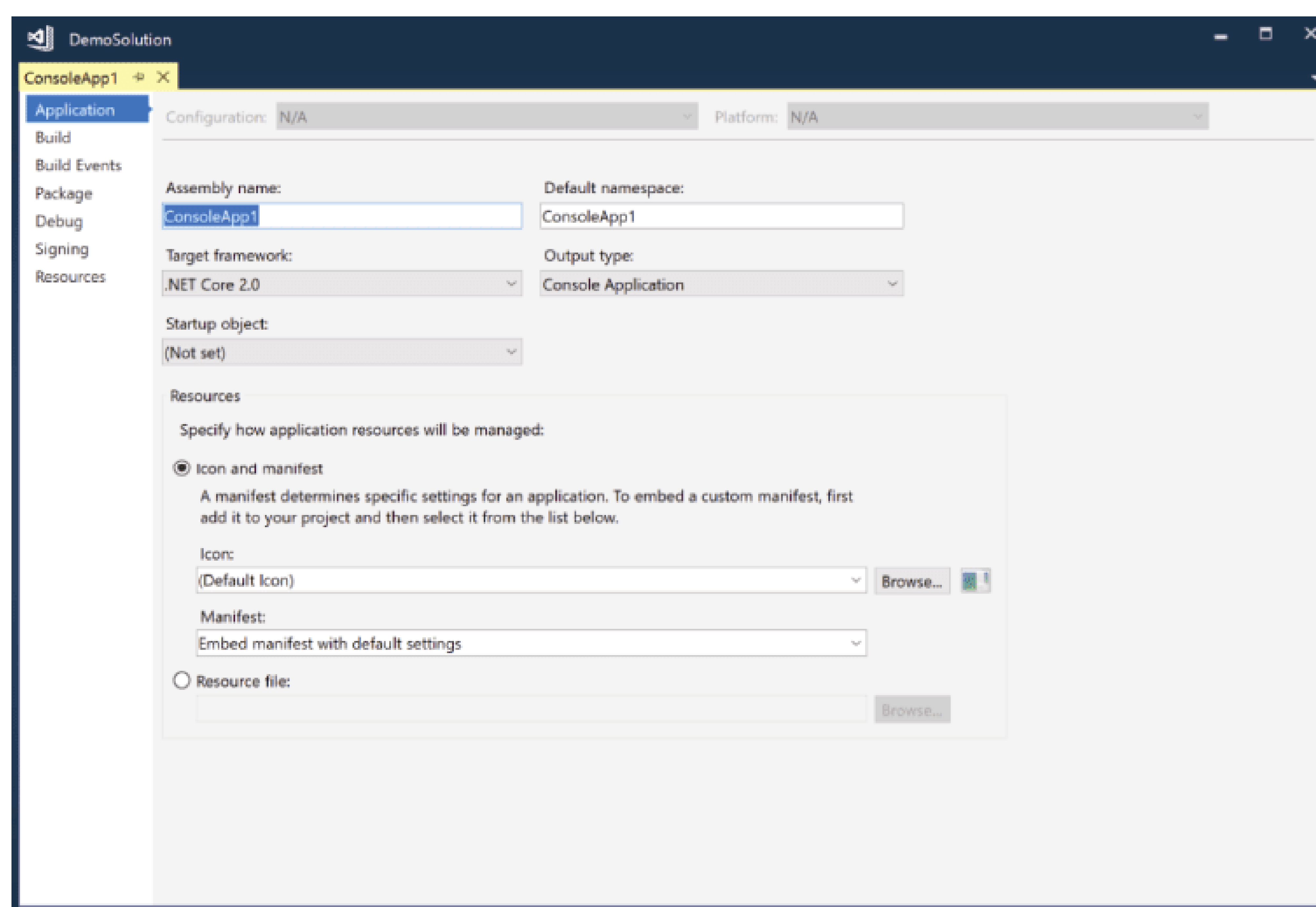



图 18-39

图 18-40 显示了 Universal Windows 应用程序的生成配置属性。注意，在对话框顶部的下拉列表中可以指定要查看的配置类型。对于 Debug 配置，编译器假定已经定义了 DEBUG 和 TRACE 预处理器符号。此外，编译器不会优化代码，而且会生成额外的调试信息，不使用.NET Native 工具链。

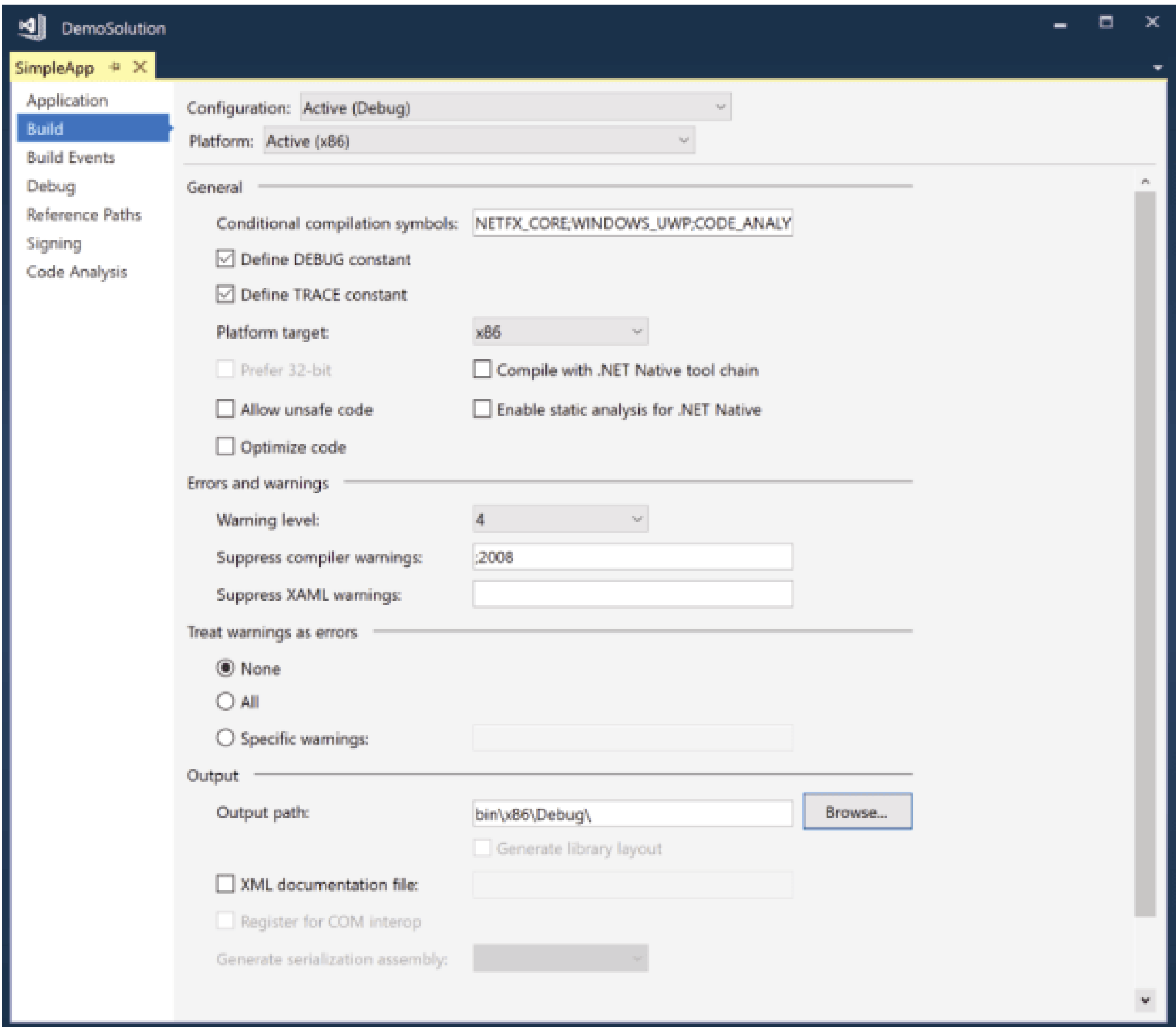



图 18-40

图 18-41 显示了.NET Core 项目的构建配置属性。同样，在调试配置中，代码没有优化，而定义了 DEBUG 和 TRACE 预处理器符号。

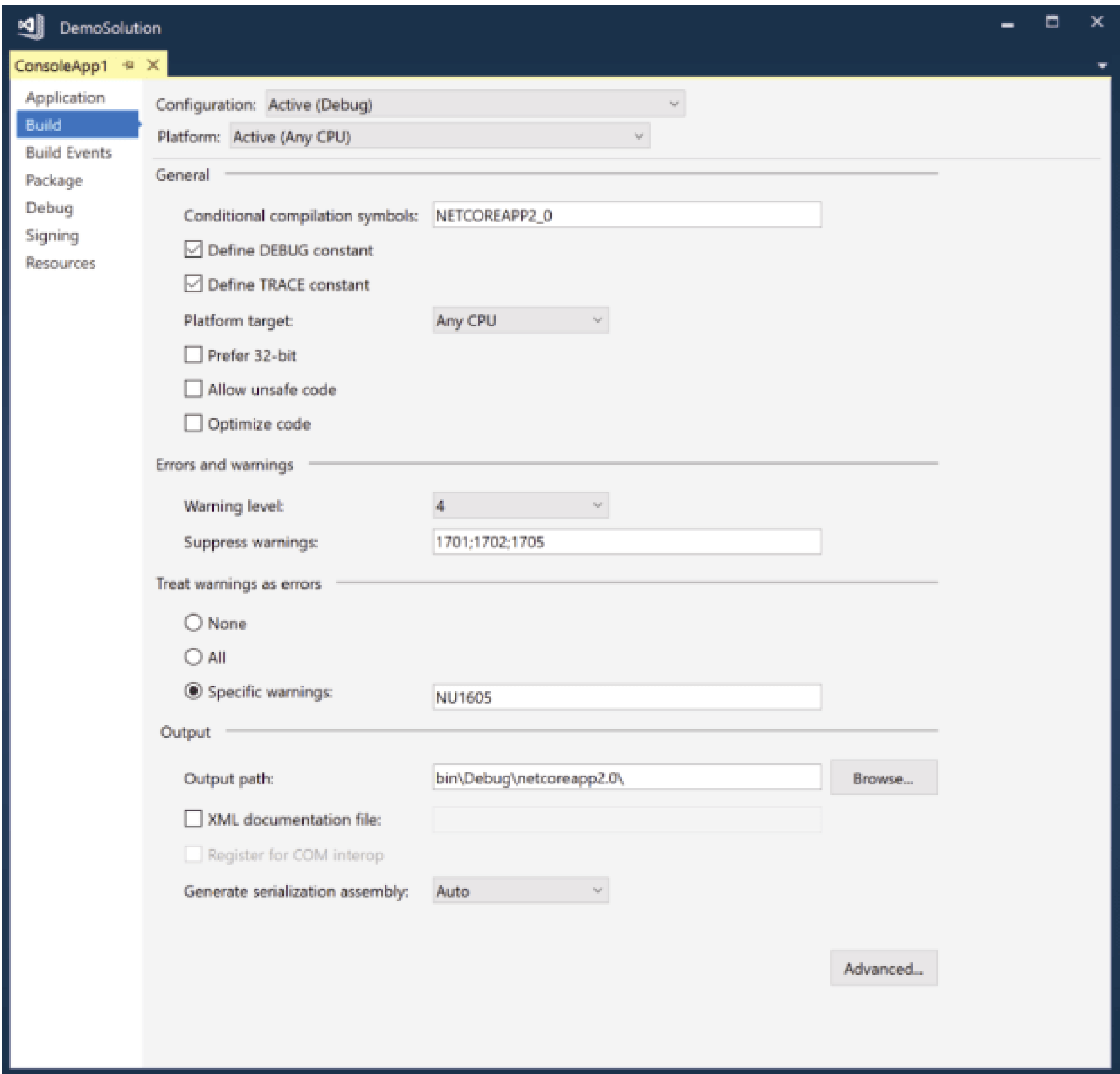



图 18-41

# 18.5 调试代码

现在，已经准备好运行和调试应用程序了。在 C#中调试应用程序与早于.NET 的语言一样，涉及的主要技巧是设置断点，使用断点检查在代码执行到特定位置时发生了什么情况。

### 18.5.1　设置断点

在 Visual Studio 中，可在实际执行的任何代码行上设置断点。最简单的方法是在代码编辑器中单击文档窗口最左边的灰色区域，或者在选中合适的行后按 F9 键。这会在该代码行上设置一个断点，当程序执行到该行时将暂停执行，并把控制权转交给调试器。与 Visual Studio 以前的版本一样，断点由代码编辑器中代码行左边的红色圆圈表示。Visual Studio 还使用不同的颜色高亮显示该行代码的文本和背景。再次单击红色圆圈将删除断点。

如果对于特定的问题，每次在特定的代码行暂停执行不足以解决该问题，则还可以设置条件断点。为此，选择 Debug | Windows | Breakpoints。在打开的对话框中，输入想要设置的断点的细节。例如，在该对话框中可以执行以下操作，如图 18-42 所示：

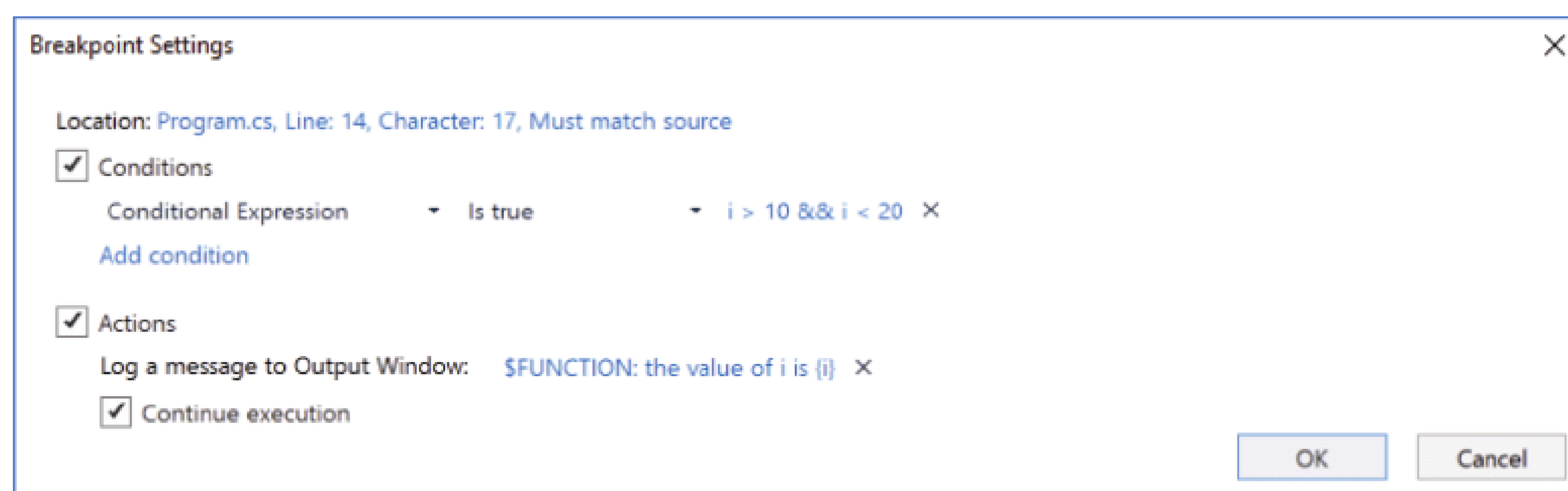



图 18-42

- 指定条件——例如，只有当条件表达式返回 true 时，才激活断点。你还可以指定遇到代码行达到一定次数时，才激活断点。
- 设置将消息记录到输出窗口并继续执行的操作。

使用该对话框还可以导出和导入断点设置，如果根据不同的调试场景想要使用不同的断点设置，那么这些选项十分有用。另外，在该对话框中还可以存储调试设置。

### 18.5.2　使用数据提示和调试器可视化工具

在断点触发后，通常想要查看变量的值。最简单的方法是在代码编辑器中，在变量名的上方悬停光标。这将弹出一个很小的数据提示框，其中显示了该变量的值。也可以展开数据提示框来查看更多细节，如图 18-43 所示。

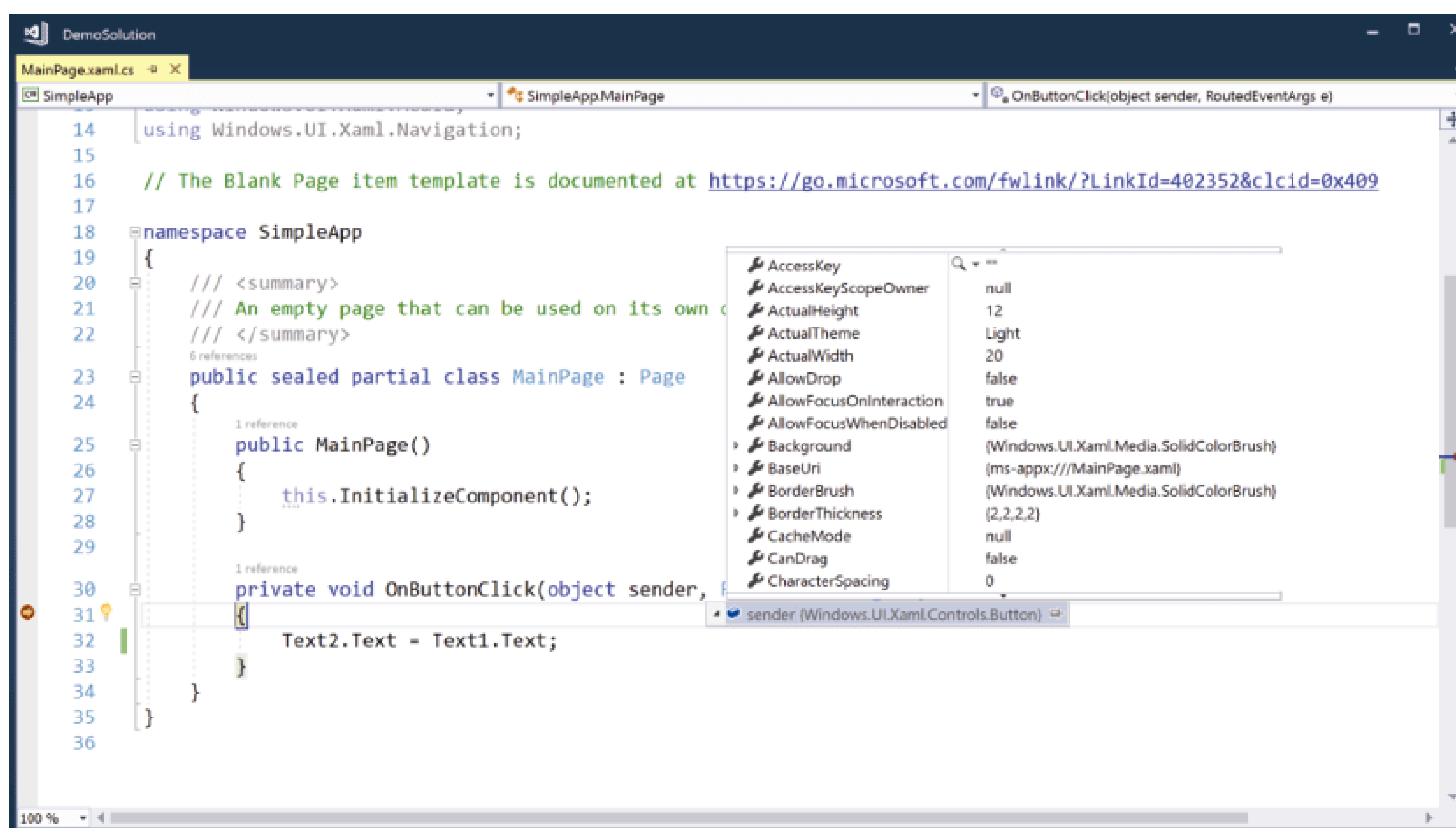



图 18-43

数据提示中的某些值会带有一个放大镜图标。单击这个放大镜图标时，根据数据的类型，会显示一个或多个使用调试器可视化工具(debugger visualizer)的选项。如图 18-44 所示的 JSON Visualizer 可显示 JSON 内容。还有其他许多可视化工具，如 HTML、XML 和 Text 可视化工具。

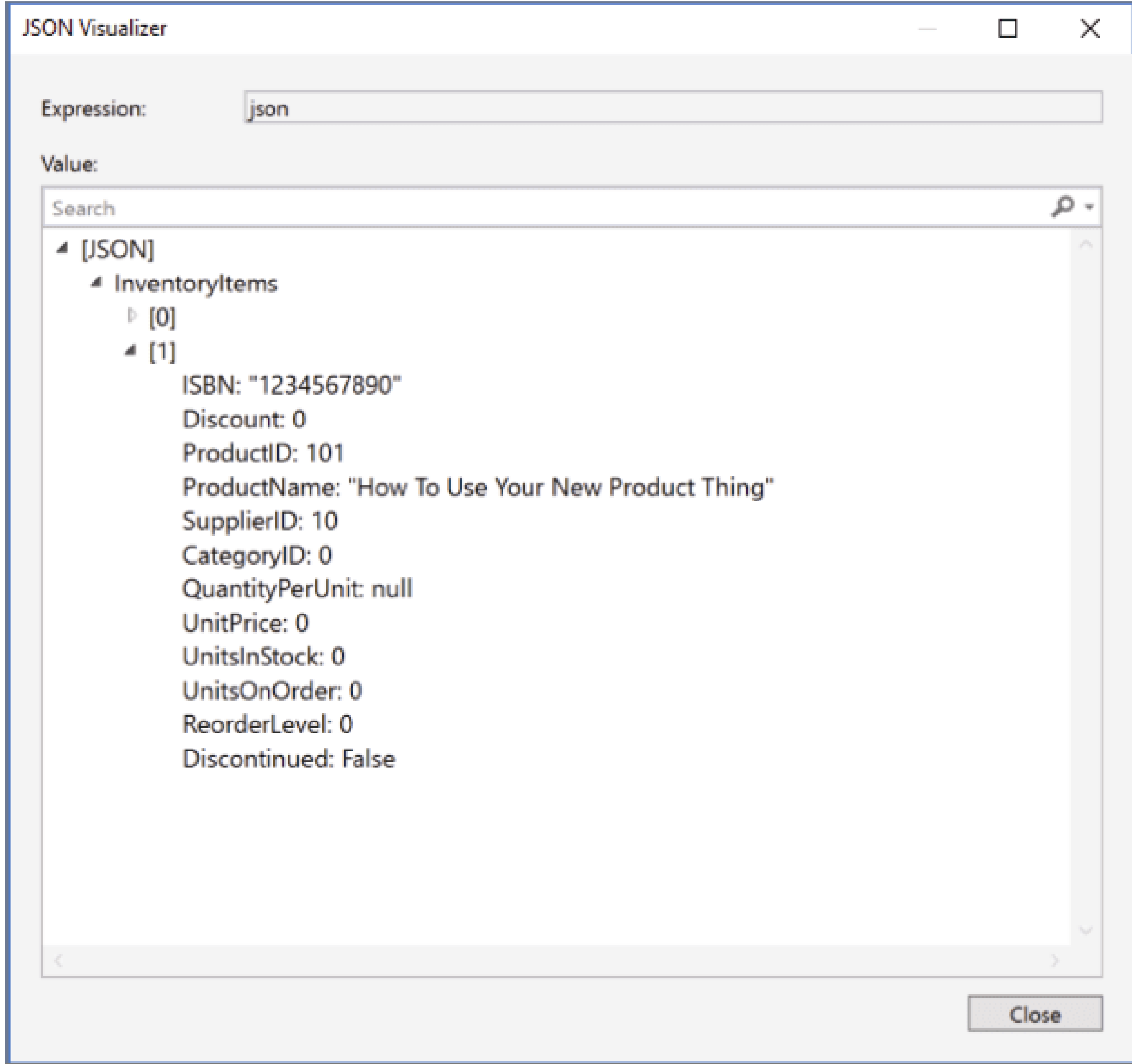



图 18-44

### 18.5.3 Live Visual Tree

Visual Studio 2017 为基于 XAML 的应用程序提供了一个新特性：Live Visual Tree。调试 UWP 和 WPF 应用程序时，可以打开 Live Visual Tree (见图 18-45)：选择 Debug | Windows | Live Visual Tree，就会在 Live Property Explorer 中打开 XAML 元素的实时树(包括其属性)。使用此窗口，可以单击 Selection 按钮，在 UI 中选择一个元素，在树中查看它的元素。在 Live Property Explorer 中，可以直接改变属性，看看这种改变在运行着的应用程序上的结果。

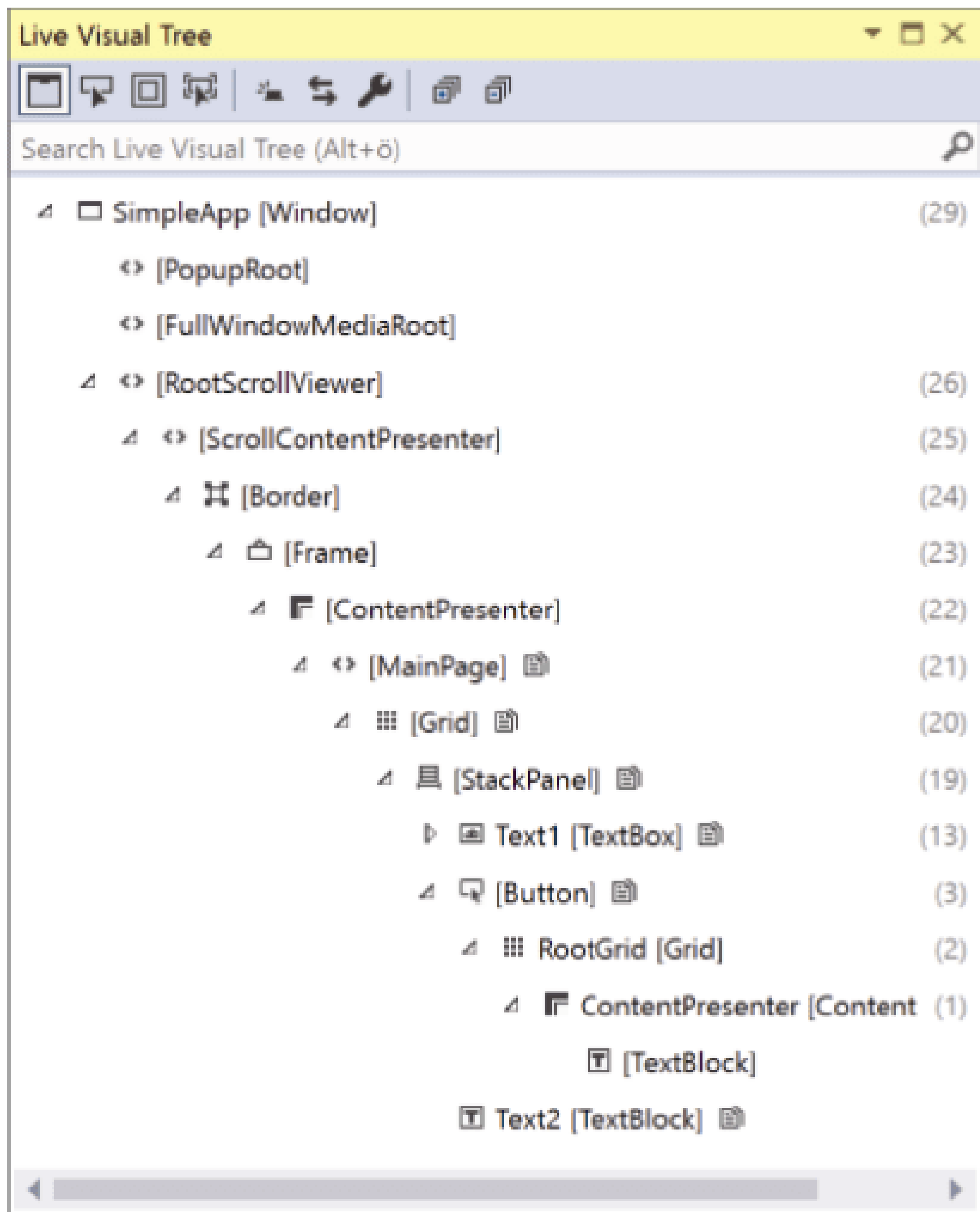



图 18-45

### 18.5.4 监视和修改变量

有时候，需要连续查看变量的值。此时，可以使用 Autos、Locals 和 Watch 窗口检查变量的内容。这 3 个窗口监视不同的变量：

- Autos——监视在程序执行过程中离断点最近的几个变量。
- Locals——监视在程序执行过程中当前断点所在方法内可访问的变量。
- Watch——监视在程序执行过程中显式指定名称的任何变量。可以把变量拖放到 Watch 窗口中。

只有当使用调试器运行程序时，这些窗口才是可见的。如果看不到它们，则选择 Debug | Windows，然后根据需要选择菜单项。考虑到可能要监视的内容过多，需要进行分组，Watch 窗口提供了 4 个不同的窗口。在这些窗口中都可以查看和修改变量的值，所以不必离开调试器就可以尝试改变程序的不同路径。图 18-46 显示了 Locals 窗口。

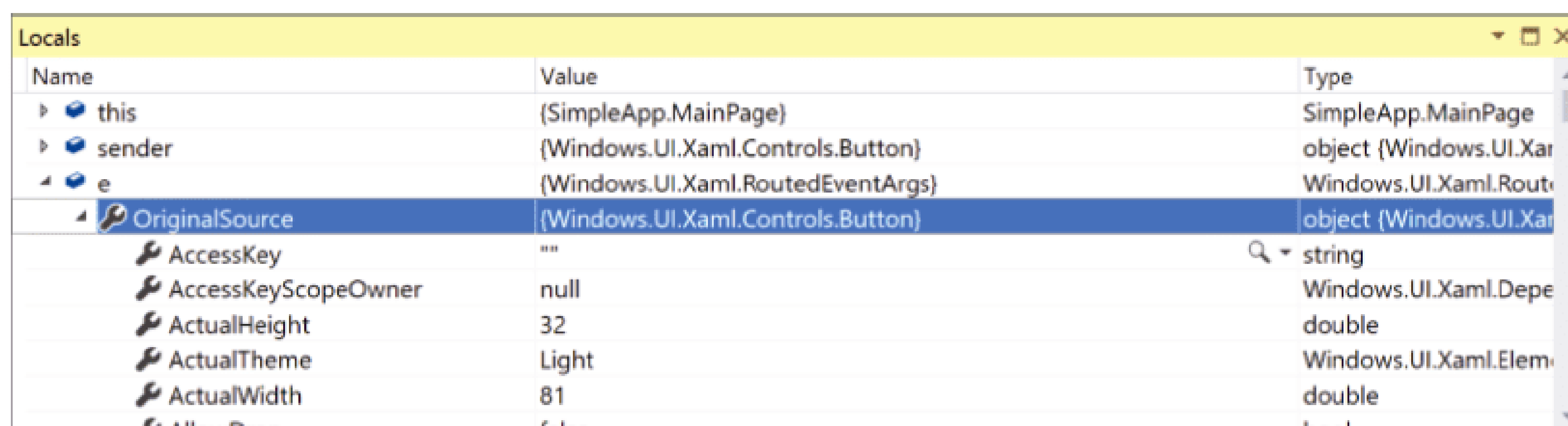



图 18-46

另外，还有一个 Immediate 窗口，虽然它与刚才讨论的其他几个窗口没有直接关系，但仍然是一个可以监视和修改变量的重要窗口。在该窗口中可以查看变量的值。还可以在此窗口中输入并运行代码，这样在调试过程中进行测试时就能够关注细节，试用方法，以及动态修改调试运行。

### 18.5.5 异常

在准备交付应用程序时，异常是很好的帮手，它们确保错误条件得到了恰当的处理。如果正确使用，异常就能够确保用户不会看到技术性或者恼人的对话框。但是，在调试应用程序时，异常就没有那么令人愉快了。它们的问题在于两个方面：

- 如果在调试过程中发生异常，通常不希望自动处理它们，特别是有时自动处理异常意味着应用程序将终止。调试器应能帮助确定为什么会发生异常。当然，如果编写了优良、健壮的防御性代码，程序将能够自动处理几乎任何事情，包括想要检测的 bug！
- 如果发生了某个异常，.NET 运行库就会尝试搜索该异常的处理程序，即使没有为该异常编写处理程序。没有找到异常时，它会终止程序。这时没有了调用栈，意味着所有的变量将超出作用域，所以将无法查看任何变量的值。

当然，可以在 catch 块中设置断点，但是这常常没有多大帮助，因为按照定义，当遇到 catch 块时，执行流已经退出了对应的 try 块。这意味着想要通过检查变量值来确定问题所在时，那些变量已经超出了作用域。甚至不能通过查看栈跟踪来找出在遇到 throw 语句时执行的方法，因为控制流已经离开了该方法。在 throw 语句处设置断点显然可以解决这个问题，但是对于防御性编码，代码中将存在许多 throw 语句。如何判断哪条 throw 语句抛出了该异常？

Visual Studio 为这种问题提供了一个很好的解决方法。可以配置调试器中断处的异常类型。这在菜单 Debug | Windows | Exception Settings 中配置。如图 18-47 所示，在该窗口中可以指定抛出异常后执行什么操作。例如，可以选择继续执行，或者停止执行并启动调试——此时程序将停止执行，调试器将在 throw 语句位置启动。

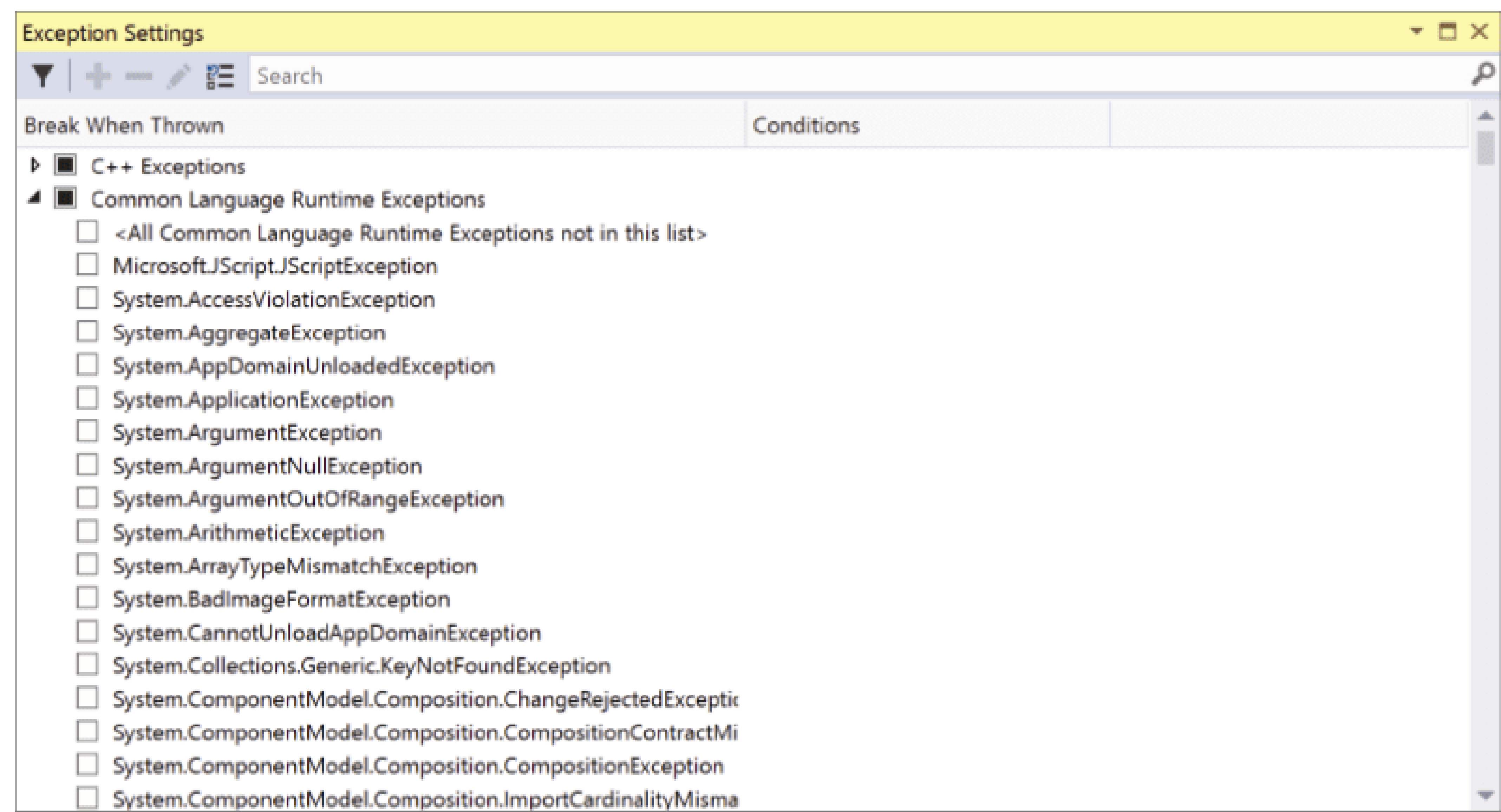



图 18-47

这个对话框的强大之处在于允许根据所抛出异常的类型选择相应的操作。例如，可以配置为在遇到.NET 基类抛出的任何异常时进入调试器，但是对于其他异常类型则不进入调试器。

Visual Studio 知道.NET 基类中的所有异常类，以及许多可在.NET 环境外抛出的异常。Visual Studio 不会自动知道用户编写的任何自定义异常类，但是可以把自定义异常类手动添加到 Visual Studio 的异常列表中，并指定哪些自定义异常类应该导致执行立即停止。为此，只需要单击 Add 按钮(在列表中选择一个顶层节点时，将启用该按钮)，并输入自定义异常类的名称即可。

## 18.5.6 多线程

Visual Studio 为调试多线程程序提供了出色的支持。在调试多线程程序时，必须理解在调试器中运行与不在调试器中运行时，程序的行为会发生变化。遇到断点时，Visual Studio 会停止程序的所有线程，所以此时有机会查看所有线程的当前状态。为了在不同的线程间切换，可以启用 Debug Location 工具栏。这个工具栏有一个针对所有进程的组合框，还有另外一个组合框，用于当前运行的应用程序的所有线程。当选择一个不同的线程时，可以看到该线程在哪一行代码暂停，以及当前可在其他线程中访问的变量。Tasks 窗口(如图 18-48 所示)显示了所有正在运行的任务，包括这些任务的状态、位置、任务名、任务使用的当前线程、所在的应用程序域以及进程标识符。该窗口还显示了不同的线程在什么时候彼此阻塞，导致死锁。

Tasks

| | ID | Status | Start Tim... | Duration... | Location | Task |
|---|---|---|---|---|---|---|
| | 1 | Deadlock | 0.000 | 173.877 | ThreadingIssues.Proc | ThreadingIssues.Pı |
| | 2 | Deadlock | 0.000 | 173.877 | ThreadingIssues.Proc | ThreadingIssues.Pı |
| | 3 | Scheduled | 0.000 | 173.877 | [Scheduled and waiti | Task.Delay |

图 18-48

图 18-49 显示了 Parallel Stacks 窗口，该窗口以分层视图的形式显示了不同的线程或任务(取决于所选项)。单击任务或线程可跳转到对应的源代码。

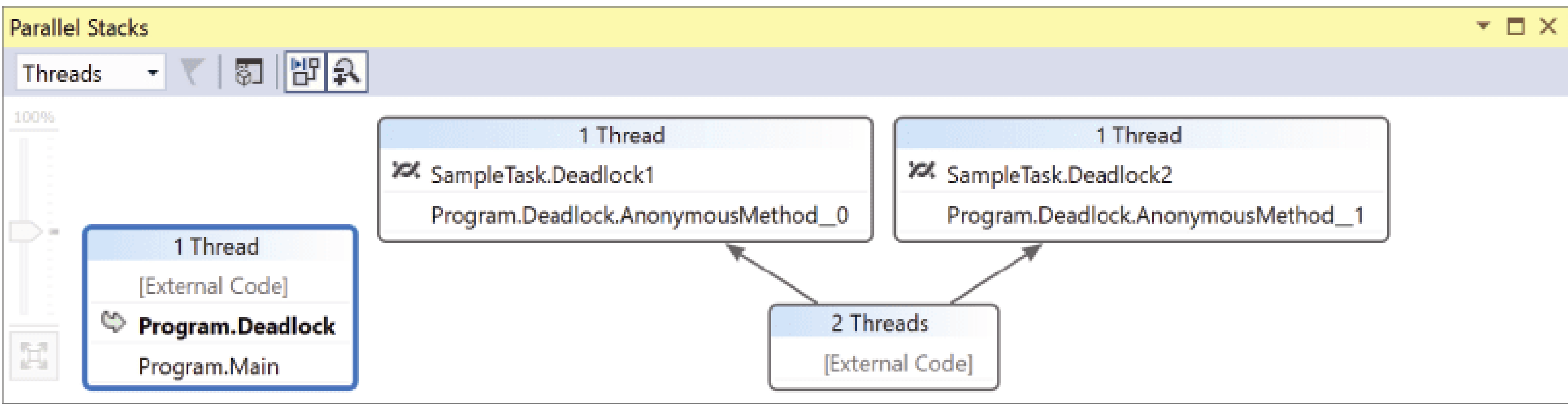



图 18-49

## 18.6　重构工具

许多开发人员在开发应用程序时首先完成功能，然后修改应用程序，使它们更易于管理和阅读。这个过程称为重构。重构过程包括修改代码来实现更好的性能和可读性，提供类型安全，以及确保应用程序符合标准的面向对象编程实践。更新应用程序时，也需要重构。

Visual Studio 2017 的 C#环境包含一组重构工具，位于 Visual Studio 菜单的 Refactoring 选项中。为了演示这些工具，在 Visual Studio 中创建一个新类 Car：

```
public class Car
{
  public string _color;
  public string _doors;

  public int Go()
  {
    int speedMph = 100;
    return speedMph;
  }
}
```

现在，假设为了进行重构，需要对代码稍作修改，将变量 color 和 door 封装到公有的.NET 属性中。Visual Studio 2017 的重构功能允许在文档窗口中简单地右击这两个属性，然后选择 Quick Actions，就会看到不同的重构选项，例如生成构造函数，来填充字段，或者输出方法 Equals 和 GetHashCode，或者封装字段，如图 18-50 所示。

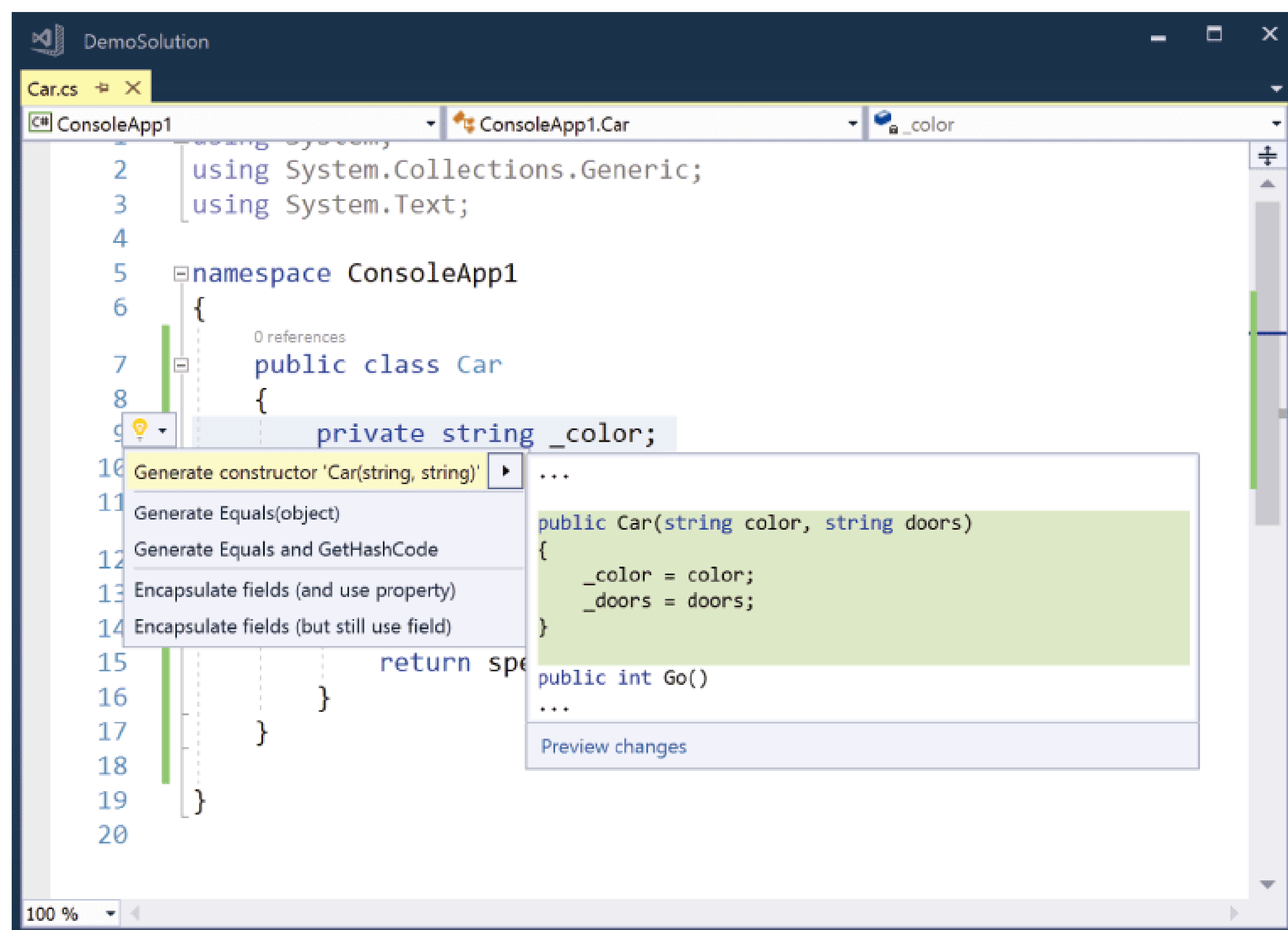



图 18-50

在该对话框中可以提供属性的名称，然后单击 Preview 链接，或者直接接受更改。选择封装字段的按钮时，代码将修改为如下所示：

```
public class Car
{
  private string _color;
  private string _doors;

  public string Color { get => _color; set => _color = value; }
  public string Doors { get => _doors; set => _doors = value; }

  public int Go()
  {
```

```
        int speedMph = 100;
        return speedMph;
    }
}
```

可以看到，使用这些向导重构代码很简单，不仅是一页的代码，重构整个应用程序的代码都一样简单。Visual Studio 的重构工具还提供了以下功能：

- 重命名方法、局部变量、字段等
- 从选定代码中提取方法
- 基于一组已有的类型成员提取接口
- 将局部变量提升为参数
- 重命名参数或修改参数的顺序

Visual Studio 2017 的重构工具是得到更整洁、可读性更好、结构更合理的代码的一种优秀方法。

## 18.7 诊断工具

Visual Studio 2017 提供了许多有用的工具来帮助分析应用程序，提前解决应用程序中可能发生的问题。本节将讨论其中的一些分析工具。

与体系结构工具类似，分析工具也在 Visual Studio 2017 企业版中可用。

为了分析应用程序的完整运行，可以使用诊断工具。诊断工具用于确定调用了什么方法、方法的调用频率、方法调用所需的时间、使用的内存量等。在 Visual Studio 2017 中，启动调试器时，会自动启动诊断工具。使用诊断工具，还可以看到 IntelliTrace(历史调试)事件。遇到一个断点后，能够查看以前的信息(如图 18-51 所示)，例如以前的断点、抛出的异常、数据库访问、ASP.NET 事件、跟踪或者用户操作(如单击按钮)。单击以前的事件时，可以查看局部变量、调用栈以及函数调用。使用这种功能时，不需要重启调试器并为发现问题前调用的方法设置断点，就可以轻松地找到问题所在。

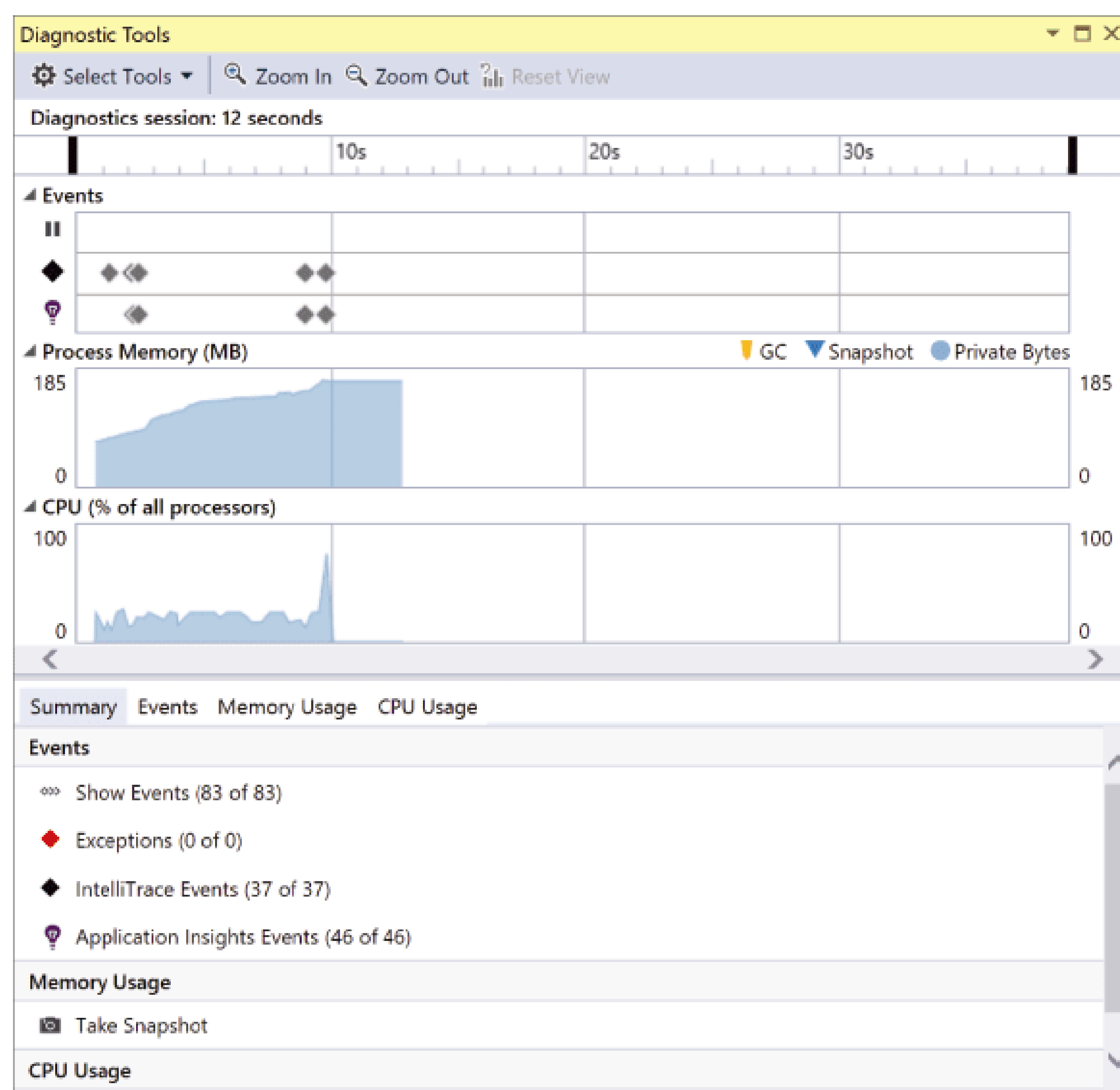



图 18-51

**注意：**

IntelliTrace 也在 Visual Studio 2017 企业版中可用。

启动诊断工具的另一种方法是通过配置文件来启动：Debug | Performance Profiler 或 Analyze | Performance Profiler。这里可以对要启动的功能进行更多的控制(见图 18-52)。根据使用的项目类型，可以使用或多或少的特性。对于 UWP 项目，可以看到应用程序的时间线、内存使用情况、CPU 使用情况和内存。其他工具包括 HTML UI 响应、JavaScript 内存、UI 分析和性能向导。

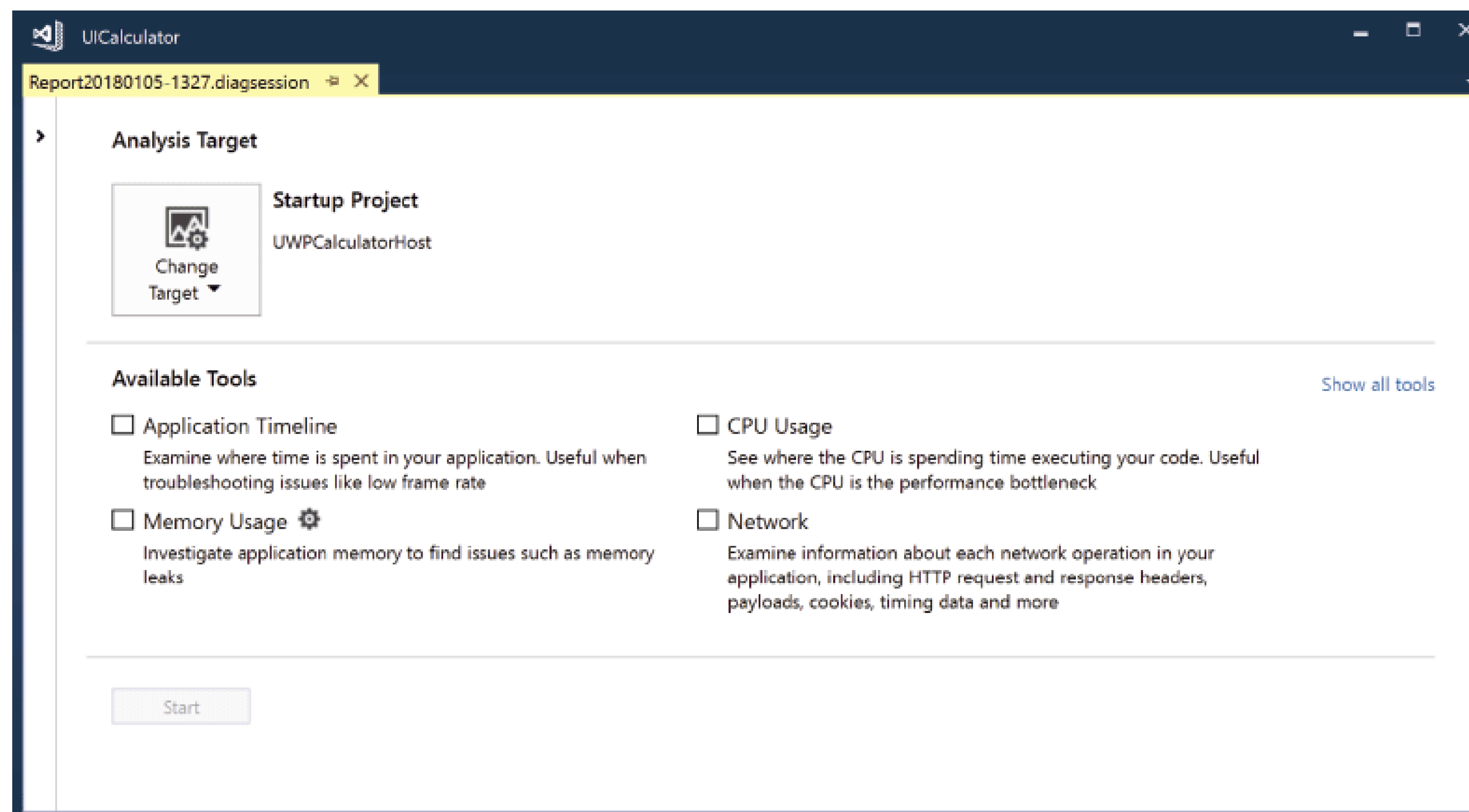



图 18-52

第一个选项 Application Timeline (见图 18-53)提供了 UI 线程的信息，以及解析、布局、渲染、I/O 和应用代码所花的时间。根据所花的最多时间，可以确定优化在哪里是有用的。

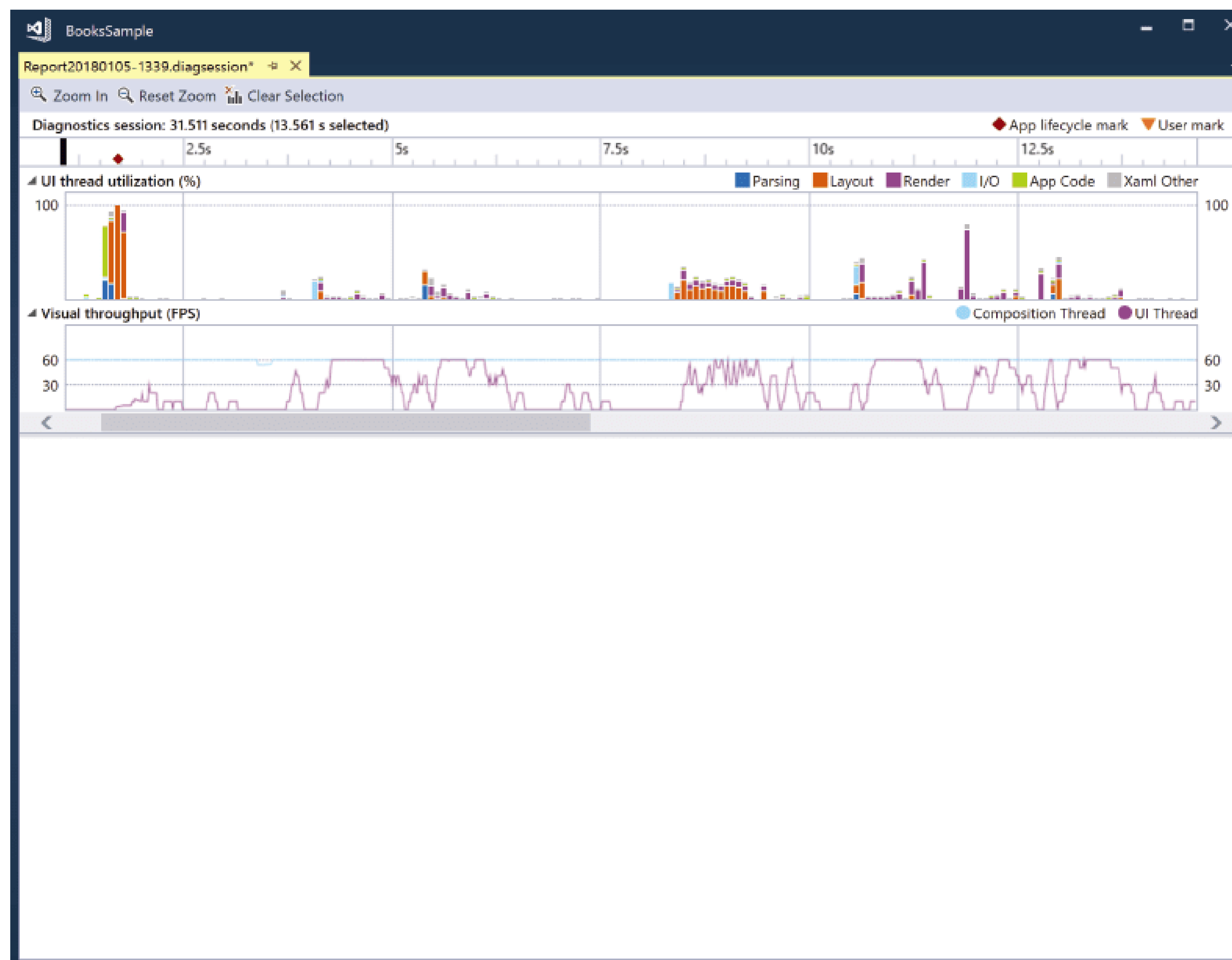



图 18-53

如果选择 CPU Usage 选项(如图 18-54 所示)，监控的开销就很小。使用此选项时，每经过固定的时间间隔就对性能信息采样。如果方法调用运行的时间很短，就可能看不到这些方法调用。但是再提一次，这个选项的优势在于开销很低。进行探查时总是应该记住，并不只是在监视应用程序的性能，也是在监视数据获取操作的性能。所以不应该同时探查全部数据，因为采样全部数据会影响得到的结果。收集关于.NET 内存分配的信息有

助于找出发生内存泄漏的地方，以及哪种类型的对象需要多少内存。资源争用数据对分析线程有帮助，能够很容易地看出不同的线程是否会彼此阻塞。

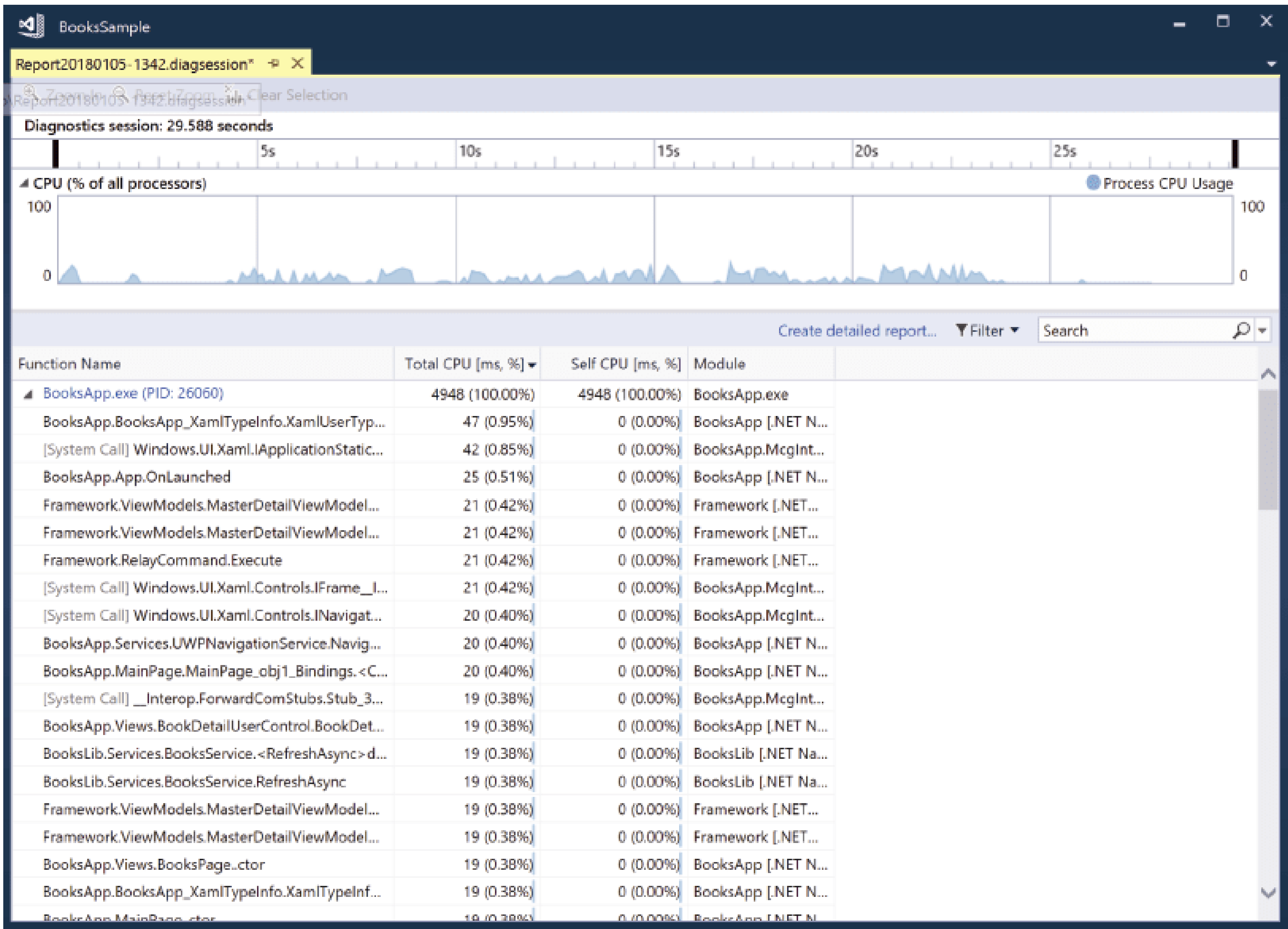



图 18-54

完成了分析运行后，图 18-55 显示了一个探查会话的摘要屏幕。从中可以看到应用程序的 CPU 使用率，说明哪些函数占用最长时间的热路径(hot path)，以及使用最多 CPU 时间的函数的排序列表。

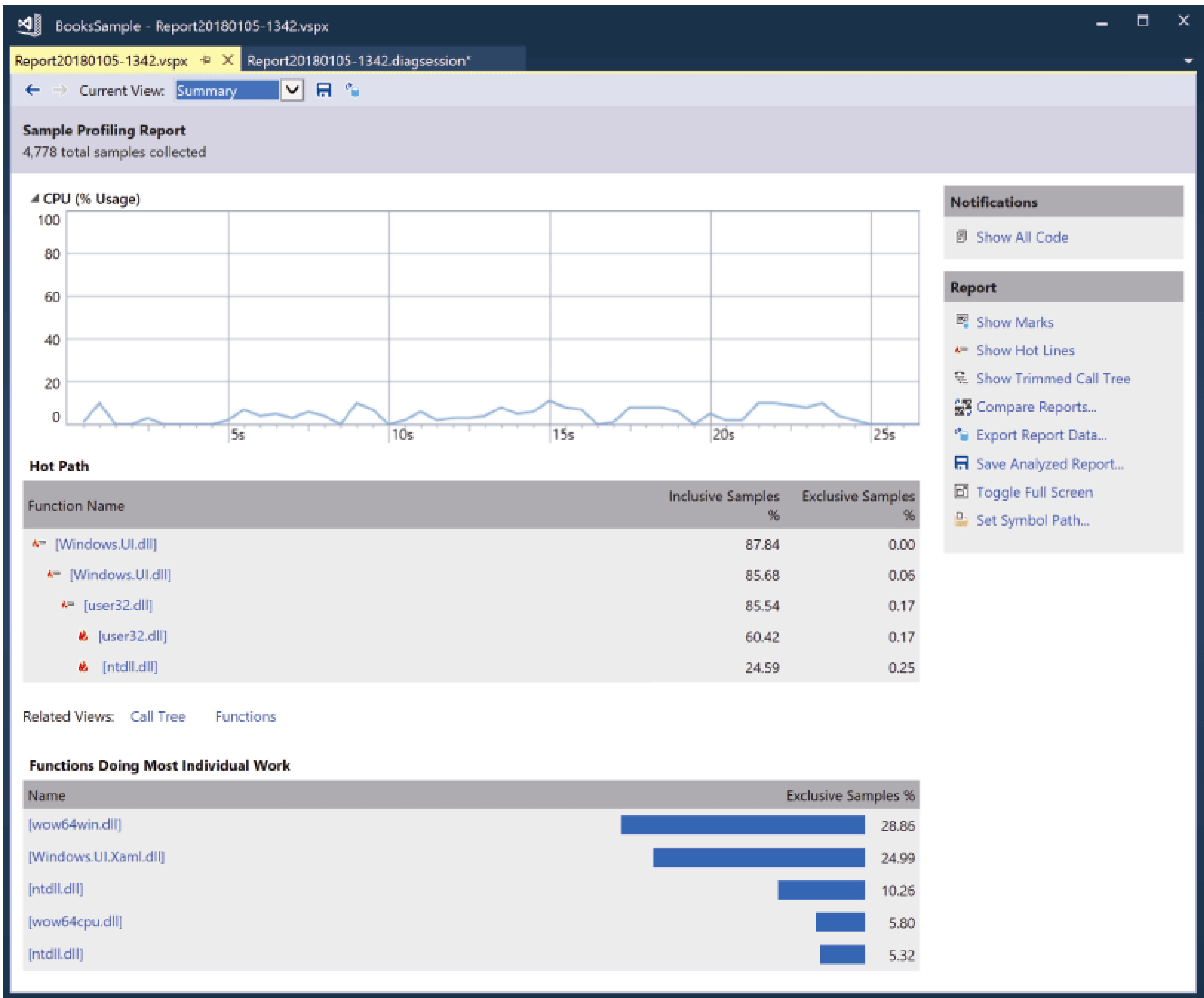



图 18-55

诊断工具还有许多屏幕，这里无法一一展示。其中有一个函数视图，允许根据函数调用次数进行排序，或者根据函数占用的时间(包括或者不包含函数调用本身)进行排序。这些信息有助于确定哪些方法的性能值得关注，而其他方法则可能因为调用得不是很频繁或者不会占用过多时间，所以不必考虑。

在函数内单击，就会显示该函数的详细信息，如图 18-56 所示。这样就可以看到调用了哪些函数，并立即开始单步调试源代码。Caller/Callee 视图也会显示函数的调用关系。

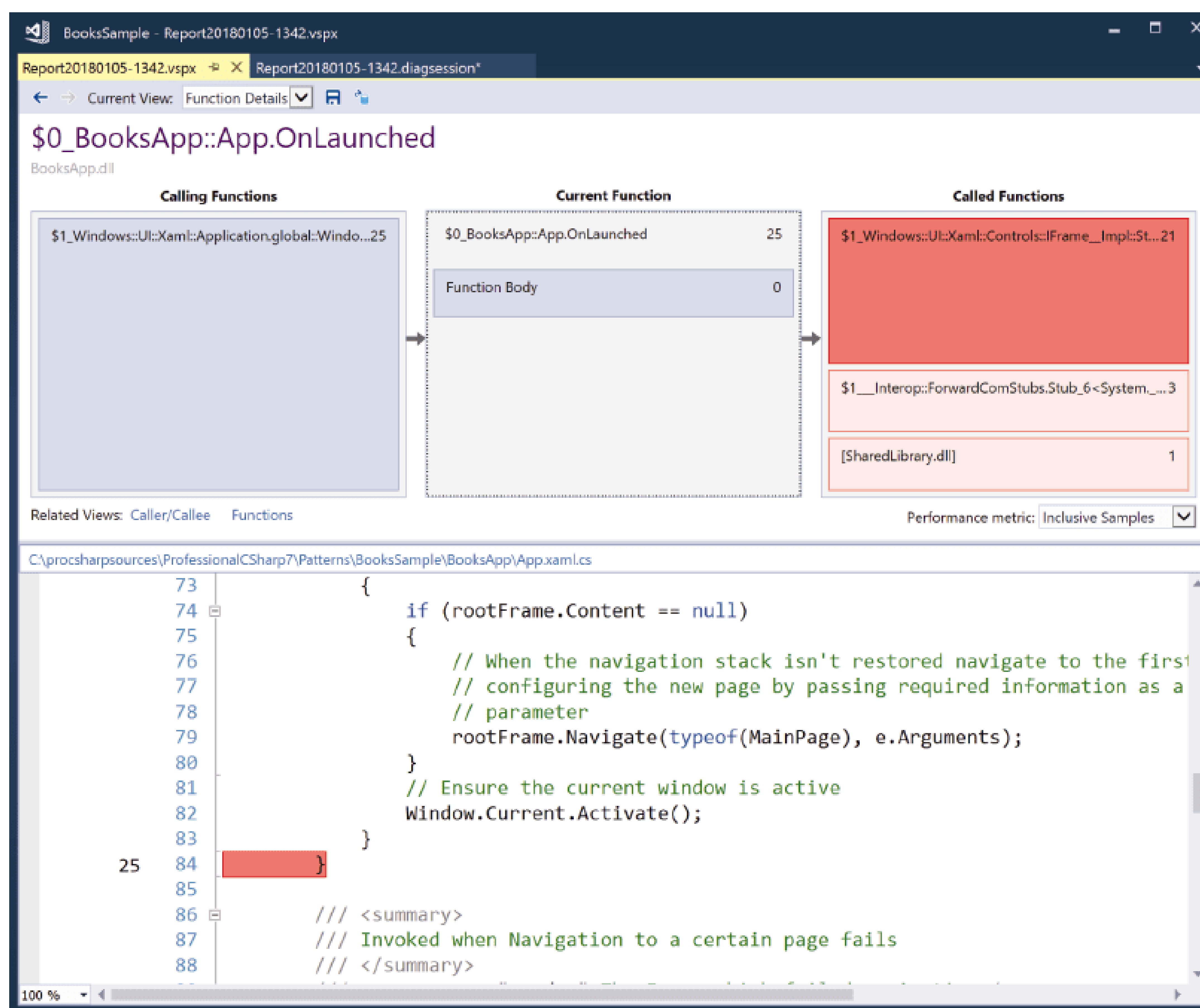



图 18-56

## 18.8　通过 Docker 创建和使用容器

Visual Studio 的一个杰出的新特性是它的 Docker 集成。因为 Docker 在.NET 领域中相对较新，值得介绍。

Docker 有什么好处？Docker 提供了虚拟化；它是一个容器技术，能够打包并部署应用程序和服务，且与所有依赖项相隔离。与虚拟机相比，它是轻量级的，因为映像可以小得多，因为一个映像是基于另一个映像的。一个 Docker 容器也不需要保留一个 CPU 内核和内存，因为它们可以共享。可以通过开发创建用于生产的 Docker 映像。它不再是，“它在我的机器上运行。”

以下是了解 Docker 的关键术语：

- **映像**——部署单元。映像是一个包，其中包括运行应用程序所需的所有依赖项(框架)和配置。映像可以从其他映像中得到。在映像创建之后，就是不可变的。
- **注册表**——存储 Docker 映像的存储区。在 Docker 的官方注册中心 https://hub.docker.com 可以找到成千上万的映像。在 https://hub.docker.com/r/microsoft/dotnet /上可以为 Linux 和 Windows 服务器拉出官方的 Microsoft .NET Core 映像。
- **容器**——一个运行的映像。容器是一个应用程序或服务的运行库环境。可以通过从同一个映像中创建多个容器实例来扩展它。容器在主机中运行。可以在 https://store.docker.com/editions/community/docker-ce-desktop-windows 上下载 Docker Community Edition for Windows，以便在 Windows 10 上驻留 Docker。Microsoft Azure 提供了一些在云中运行 Docker 映像的产品。

- Dockerfile——一个文本文件，其中包含构建 Docker 映像的指令。可以使用 Docker 命令行来处理 dockerfile。

要了解 Docker，最好尝试一下。

### 18.8.1 Docker 简介

在安装 Docker Community Edition 之后，就可以使用 Docker 命令来提取和运行 Docker 容器。开始的第一个命令是 Docker 版本的“Hello World!”。在命令行中，只需要调用 docker run hello-world。第一次启动此操作时，可以看到如下输出所示的结果。因为 Docker 映像 hello-world 尚未在本地安装，所以 Docker 映像从 Docker 注册表中提取并启动。下载成功后，会显示 Hello from Docker!：

```
> docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
ca4f61b1923c: Pull complete
Digest: sha256:ca0eeb6fb05351dfc8759c20733c91def84cb8007aa89a5bf606bc8b315b9fc7
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
```

第二次调用该命令时，因为 Docker 映像已经在本地下载，所以会立即得到结果：

```
> docker run hello-world
Hello from Docker!
```

### 18.8.2 在 Docker 容器中运行 ASP .NET Core

接下来，从 Microsoft 获取 Docker 容器，创建并运行 ASP.NET Core 应用程序。

要获取并运行容器，可以使用以下选项启动命令 docker run：

```
> docker run -p 8000:80 -e "ASPNETCORE_URLS=http://+:80" -it --rm microsoft/dotnet
```

选项-p 将端口 80 从容器映射到容器外的 8000 端口。在本地系统上，端口 80 可能已经被 Web 服务器占用了。选项-e 在容器内设置环境变量。通过设置环境变量 ASPNETCORE_URLS，可以指定端口号，它是托管 ASP.NET Core 的 Kestrel 服务器监听的端口。选项-it 启动容器与终端交互；因此，当命令完成时，在容器的命令提示符中键入。选项--rm 删除在再次下载容器之前已经存在的容器。命令的最后一部分指定从中心提取的映像并启动。microsoft/dotnet 映像包含了带有命令行工具的.NET Core SDK。

**注意：**

microsoft/dotnet 使用已发布的 SDK 获取最新的 microsoft/dotnet Docker 映像。通过在名称中添加标记，可以获取特定的版本，并使用可以用于生产的运行库获取一个版本。microsoft/dotnet:<version>-sdk 检索 *.NET Core SDK*，microsoft/dotnet:<version>- runtime 检索运行库的映像，以及 microsoft/dotnet:< version>-runtime-deps 检索一个较小的映像，其中不包含运行库，但包含了托管自包含应用程序所需的本地二进制文件。自包含应用程序在第 1 章中解释了。有关映像的详细信息可以在 https://hub.docker.com/r/microsoft/dotnet/上找到。

启动命令时，映像从 Docker 集中提取并启动，然后就可以进入 Docker 容器。下面创建一个 ASP.NET Core MVC 应用程序，并用以下命令启动它：

```
# mkdir websample
# cd websample
# dotnet new mvc
# dotnet restore
# dotnet build
# dotnet run
```

现在可以从容器外部的浏览器中访问 http://localhost:8000，并访问容器中的 ASP.NET Core 网站。从 Docker 容器的外部可以使用如下命令：

```
> docker images
```

查看所有已下载的映像，使用如下命令：

```
> docker container list
```

查看正在运行的活动容器。这是一个简化符号：

```
> docker ps
```

### 18.8.3　创建 Dockerfile

对于目前为止所使用的命令，我们已经了解了如何提取和运行 Docker 容器。接下来创建一个运行 ASP.NET Core 应用程序的自定义 Docker 镜像。这次，要创建一个发布版本，并构建一个发布包。

首先，使用 ASP.NET Core CLI 创建 ASP.NET Core MVC Web 应用程序。给 dotnet build 命令传递-cRelease，会生成发布代码。发布 dotnet publish 命令的选项-o ./Publish 将在 Publish 子目录中写入用于发布的输出。

```
> mkdir DockerSample
> cd DockerSample
> dotnet new mvc
> dotnet restore
> dotnet build -c Release
> dotnet publish -c Release -o ./Publish
Next
```

接下来，在命令提示符的当前目录中，创建一个与 Web 应用程序相同的 dockerfile(代码文件 DockerSample/dockerfile)：

```
FROM microsoft/aspnetcore
WORKDIR /app
COPY ./publish .
RUN dir .
ENTRYPOINT [ "dotnet", "/app/DockerSample.dll" ]
RUN echo 'completed building image'
```

**警告：**

在 Linux 上运行 Docker 时，注意文件名区分大小写。在 Linux 中，文件 dockersample.dll 不同于 DockerSample.dll。

下面讨论每个命令。dockerfile 从一个 FROM 开始。新构建的映像基于 microsoft/aspnetcore 映像。此映像只包含运行库。对于构建映像，可以使用 microsoft/aspnetcore-build。microsoft/aspnetcore 本身基于 microsoft/dotnet。microsoft/aspnetcore 映像包含一组用于 ASP.NET Core 库的本机映像，因此第一次运行库没有花费时间来编译这些库。

```
FROM microsoft/aspnetcore
```

下一个命令在要创建的映像中把工作目录设置为/app 文件夹：

```
WORKDIR /app
```

COPY 命令把本地./Publish 目录的内容复制到映像中的当前目录——之前定义的工作目录：

```
COPY ./publish .
```

下一个命令只显示当前目录中的文件，以查看复制是否成功：

```
RUN dir .
```

ENTRYPOINT 定义的命令应该在通过参数运行容器时启动。在运行期间，ASP.NET Core 应用程序用 dotnet 和 DLL 的名称启动：

```
ENTRYPOINT [ "dotnet", "/app/dockersample.dll" ]
```

使用 dockerfile，可以通过命令 docker build 构建一个 docker 映像。选项-t 给映像起一个名字。最后的.命令定义了应该搜索 dockerfile 的目录。

```
> docker build -t mysampleapp .
```

**注意：**
在 docker build 命令的选项-t 中，可以标记映像，例如，使用带有版本号的 docker build -t mysampleapp:1.4。

在成功构建之后，可以运行 Docker 映像，并将 80 端口从容器内映射到容器外的 8002 端口：

```
> docker run -p 8002:80 mysampleapp
```

使用生产代码和预编译的 ASP.NET Core 映像时，如果通过浏览器访问 ASP.NET Core Web 应用程序，会体验到快速响应。

### 18.8.4 使用 Visual Studio

现在进入 Visual Studio 2017。Docker 的体验是如何集成的？使用项目模板 Web Application (Model-View-Controller)创建一个 Web 应用程序。这个项目称为 WebAppWithVS。这个名称将在生成的 dockerfile 中。使用第一个对话框，可以选择一个复选框并在 Linux 或 Windows 容器之间选择，从而启用 Docker 支持(见图 18-57)。取消选中这个选项。以后很容易添加 Docker 支持。

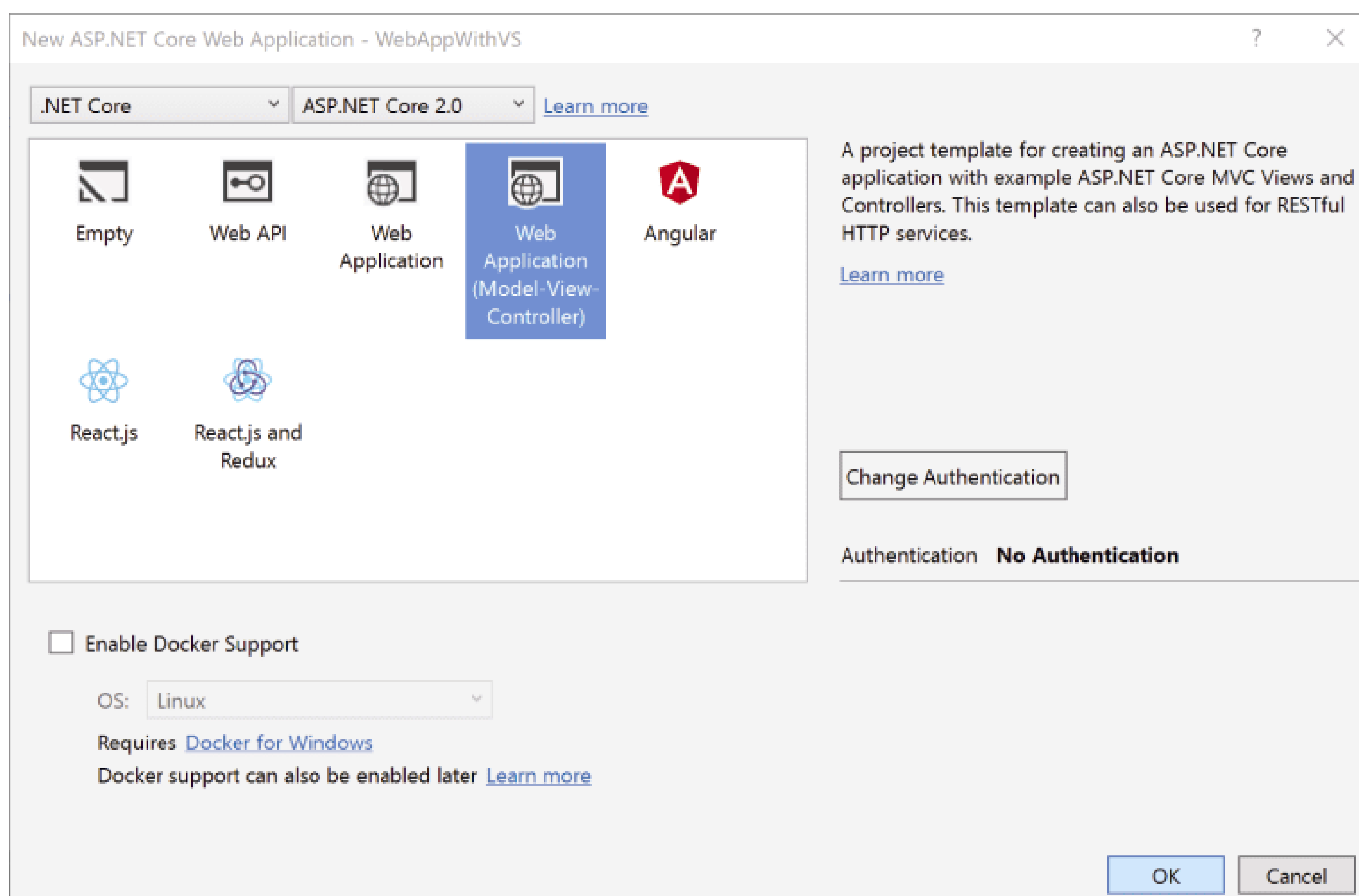



图 18-57

使用项目模板时，将创建与使用命令行时相同的文件。为了给现有的项目添加 Docker 支持，可以选择 Project | Docker Support。在图 18-58 所示的对话框中，选择目标 OS。下面在 Linux 上驻留它。

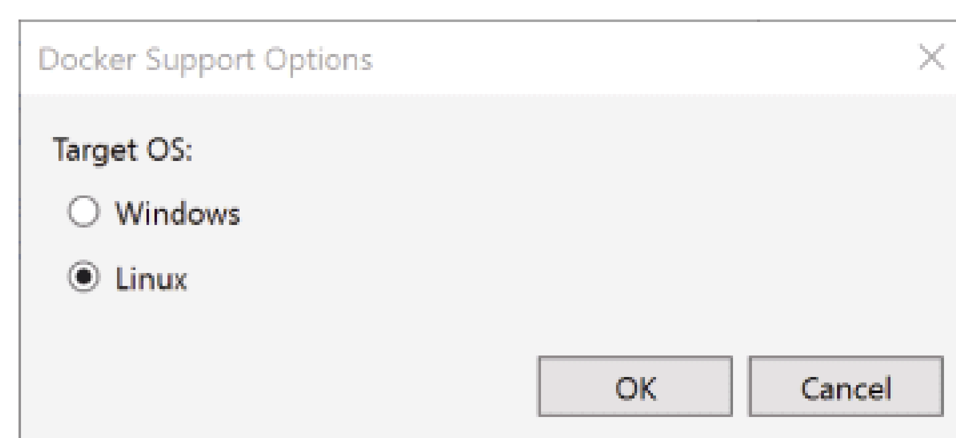



图 18-58

现在得到了项目的 dockerfile。这个 dockerfile 包含多个 FROM，用于多阶段 docker 构建。这里，根据调试或发布构建，创建多个映像。下面通过多个步骤来完成这个文件。第一个 FROM 基于映像 microsoft/aspnetcore:2.0，它定义为 AS base，以后使用名称 base 引用它。在第一个映像定义中，工作目录设置为/app，指定了端口 80(代码文件 WebAppWithVS/Dockerfile)：

```
FROM microsoft/aspnetcore:2.0 AS base
WORKDIR /app
EXPOSE 80
```

第二个 FROM 完全独立于第一个 FROM。这个 FROM 定义了映像基础 aspnetcore-build:2.0。带有 build 标记的映像是 ASP.NET Core 图像，包括 SDK。工作目录现在是/src。接下来，使用两个 COPY 命令将解决方案和项目文件复制到 rc 目录中。运行 dotnet restore 恢复所有的 NuGet 包，下一个 COPY 命令把完整的文件夹及其子目录复制到 src 目录中。在运行 dotnet build 之前，新的工作目录设置为 Web 应用程序的文件夹。

```
FROM microsoft/aspnetcore-build:2.0 AS build
WORKDIR /src
COPY *.sln ./
COPY WebAppWithVS/WebAppWithVS.csproj WebAppWithVS/
RUN dotnet restore
COPY . .
WORKDIR /src/WebAppWithVS
RUN dotnet build -c Release -o /app
```

下一个映像定义使用先前配置的构建映像：FROM build AS publish。新名称是 publish，其中，调用.NET 命令 dotnet publish，以创建/app 子目录中的发布文件：

```
FROM build AS publish
RUN dotnet publish -c Release -o /app
```

最后一个 FROM 定义了一个名为 final 的映像，它基于已定义的第一个映像：base 映像。工作目录现在是/app。COPY 命令将/app 目录从先前定义的 publish 映像(发布版本放在/app 目录中)复制到工作目录。最后，容器的入口点设置为 dotnet 命令：

```
FROM base AS final
WORKDIR /app
COPY --from=publish /app .
ENTRYPOINT ["dotnet", "WebAppWithVS.dll"]
```

有了 Dockerfile 之后，只需要调用前面所示的 Docker 命令来创建所需的映像。但是，除了使用 Docker 命令之外，还可以使用 *Docker Compose*，这是一个运行多个 Docker 应用程序的工具。对于这个工具，需要一个 YAML 文件来定义组成应用程序的服务。从服务中，可以引用 dockerfiles 构建映像。

通过 Visual Studio 将 Docker 支持添加到项目中时，可以在解决方案中找到另一个项目：docker -compose。docker-compose 项目包含的文件使用了 YAML 语法和 yml 文件扩展名。

**注意：**
YAML (YAML Ain't 标记语言)是一种语法，它不像 XML 那样复杂。YAML 在语法中使用了空白。

文件 docker-compose.yml 是自动创建的。在 Docker Compose 的版本号之后，services:将列出已构建的服务。服务名称 webappwithvs 需要匹配在 CSPROJ 配置文件中定义的<DockerServiceName>元素中的值。CSPROJ 文件还列出了用来触发这个项目的 Microsoft.Docker.Sdk。所创建的映像名称由 image 指定。可以更改这个映像名称。映像名称使用 dev(用于调试构建)和 latest(用于发布构建)标记(配置文件 WebAppWithVS/docker-compose/docker-compose.yml)：

```
version: '3'

services:
  webappwithvs:
    image: webappwithvs
    build:
      context: .
      dockerfile: WebApWithVS/Dockerfile
```

通过调试和发布构建来构建和运行项目，现在可以使用 webappwithvs:dev 和 webappwithvs:latest 创建和运行映像。虽然这已经是 Visual Studio 自动构建映像的一个很好的特性，但是最好的特性是设置断点、启动调试器，从 Windows 机器上运行的 Visual Studio 调试以及运行在 Linux 上的 Docker 映像——都在同一个系统上！

## 18.9　小结

本章探讨了.NET 环境中最重要的编程工具之一：Visual Studio 2017。大部分内容都在讲解这个工具如何简化 C#编程。

本章讨论了各种项目模板、Solution Explorer 和编辑器的各种特性。还调试了代码，重构了操作，学习了 Docker 的新集成。

本书的第一部分以这一章结束。下一章将开始深入到.NET 标准库，介绍库、程序集和 NuGet 包。

第 II 部分

# .NET Core 与 Windows Runtime

- 第 19 章　库、程序集、包和 NuGet
- 第 20 章　依赖注入
- 第 21 章　任务和并行编程
- 第 22 章　文件和流
- 第 23 章　网络
- 第 24 章　安全性
- 第 25 章　ADO.NET 和事务
- 第 26 章　Entity Framework Core
- 第 27 章　本地化
- 第 28 章　测试
- 第 29 章　跟踪、日志和分析

# 第19章

# 库、程序集、包和 NuGet

**本章要点**

- 库、程序集和包之间的差异
- 创建库
- 使用.NET 标准
- 使用共享项目
- 创建 NuGet 包

**本章源代码下载：**

打开 www.wrox.com 的 Download Code 选项卡可下载本章源代码。源代码也可以在 Libraries 目录的 https://github.com/ProfessionalCSharp/ProfessionalCSharp7 中找到。本章代码分为以下几个主要的示例文件：

- UsingLibs
- UsingLegacyLibs
- UsingASharedProject
- CreateNuGet

## 19.1 库的地狱

库可以在多个应用程序中重用代码。在 Windows 中，库有很长的历史，而构建原则通过更新的技术走向了不同的方向。在.NET 之前，动态链接库(Dynamic Link Library，DLL)可以在不同的应用程序之间共享。这些 DLL 已安装在共享目录中。这些库在同一个系统上不可能有多个版本，但它们应该是向上兼容的。当然，情况并非总是如此。此外，应用程序的安装也存在一些问题，例如没有关注指导方针，用旧版本替代共享库。这就是 DLL 地狱。

.NET 试图用程序集解决这个问题。程序集是可以共享的库。除了正常的 DLL 之外，程序集还包含可扩展的元数据，以及关于库和版本号的信息，并且可以在全局程序集缓存中并排安装多个版本。微软试图解决版本问题，但这又增加了一层复杂性。

假设使用的是应用程序 X 中的库 A 和库 B(参见图 19-1)。应用程序 X 引用库 A 的 1.1 版本和库 B 的 1.0 版本。问题是库 B 也引用了库 A，但是它引用了另一个版本——1.0。一个进程只能加载库的一个版本。那么，在

进程中加载哪个版本的库 A？如果库 B 在库 A 之前使用，那么版本 1.0 就会胜出。当应用程序 X 需要使用库 A 时，这就是一个大问题。

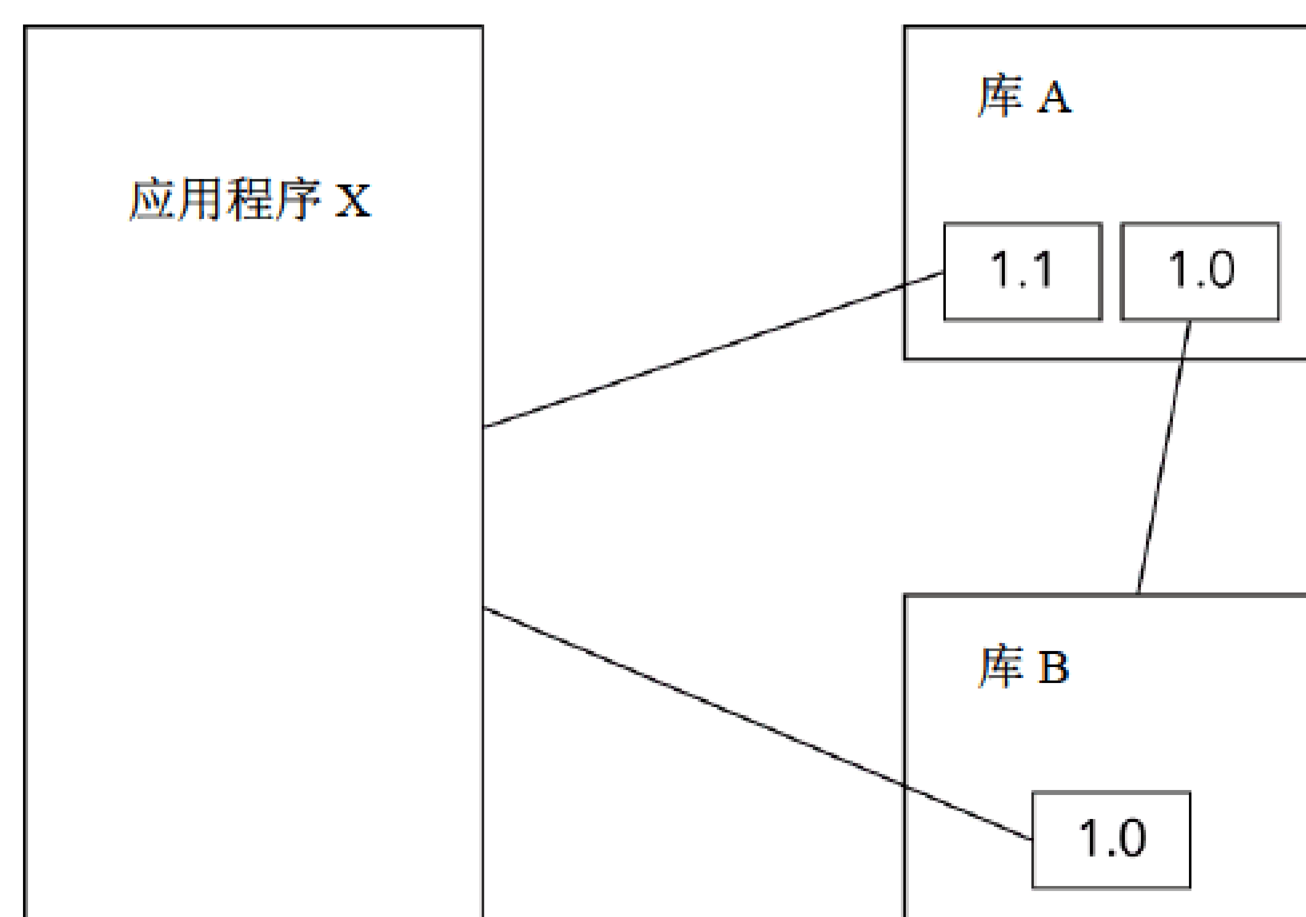



图 19-1

为了避免这个问题，可以配置程序集重定向。可以为应用程序 X 定义一个程序集重定向，以便加载库 A 的版本 1.1，然后库 B 需要使用库 A 的版本 1.1。如果库 A 是向上兼容的，这就不应该有问题。

当然，兼容性并不总是存在的，问题可能更复杂。组件的发布者可以创建一个发布者策略来定义库本身的重定向。这个重定向可以由应用程序重写。这里面有很多复杂的东西，导致了程序集的地狱。

**注意：**

*在.NET Core 中，并没有像.NET Framework 那样的程序集全局共享。只有.NET 运行库可以在不同的应用程序之间共享。*

NuGet 包在库中添加了另一个抽象层。NuGet 包可以包含一个或多个程序集的多个版本，以及其他内容，例如程序集重定向的自动配置。

不需要等待新的.NET Framework 版本，而可以通过 NuGet 包添加功能，这样可以更快地更新包。NuGet 包是一款很棒的运载工具。一些库，例如 Entity Framework，已切换到 NuGet 上，它提供的更新比.NET Framework 更快。

然而，NuGet 也存在一些问题。经常在项目中添加 NuGet 包会失败。NuGet 包可能与项目不兼容。当添加包成功时，包可能会在项目中进行一些不正确的配置，例如错误的绑定重定向。这就导致了“NuGet 包的地狱”。DLL 的问题转移到不同的抽象层，并且确实是不同的。在 NuGet 的新版本和升级版本中，微软尝试用 NuGet 解决问题。

.NET Core 体系结构中的方向也发生了变化。对于.NET Core，包的粒度更细。例如，在.NET Framework 中，Console 类位于 mscorlib 程序集中，这是每个.NET Framework 应用程序都需要的程序集。当然，并不是每个.NET 应用程序都需要 Console 类。在.NET Core 1.0 中有一个单独的包 System.Console，其中包含 Console 类和一些相关类。其目标是使更新更容易，并选择真正需要的包。在.NET Core 1.0 的一些 Beta 版本中，项目文件包含了大量的包，这并没有使开发变得更容易。在.NET Core 1.0 发布之前，微软引入了元包(或称为引用包)。元包不包括代码，而包括其他包的列表。

.NET Core 2.0 包含了另一种简化。用.NET Core 1.1 创建“Hello, World!”控制台应用程序时，生成的 project.asset.json 文件的大小为 313 KB。这个文件(在 obj 目录中)显示了依赖树。在.NET Core 2.0 中，由于包更大，引用却较少，文件大小减少到 33 KB。

本章将介绍程序集和 NuGet 包的细节，解释如何使用.NET 标准库共享代码，并解释与 Windows 运行库组件的不同之处。

## 19.2　程序集

程序集是包含额外元数据的库或可执行文件。使用.NET Core，包含 Main()方法的应用程序会创建为具有文件扩展名.dll 的库。这个 DLL 需要一个宿主进程来加载这个库，为此可以使用 dotnet run，或者在运行库环境中使用 dotnet。使用.NET Core 创建独立的应用程序时，会为每个平台创建不同的可执行文件来加载库。

下面看看一个简单的“Hello, World!” 控制台应用程序，它使用如下命令在目录 ConsoleApp 中创建：

```
> dotnet new console
```

在构建应用程序之后，DLL 可以在 bin/debug/netcoreapp2.0 目录中找到。netcoreapp2.0 目录依赖于 csproj 项目文件中列出的目标框架。

可以使用 ildasm.exe (IL 反汇编程序)命令行实用程序读取程序集信息。ildasm.exe 显示了程序集及其成员的类型，和附加的元数据，如图 19-2 所示。

图 19-2

单击MANIFEST(参见图19-2)，为程序集打开元数据信息，如图19-3所示。该程序集的名称是ConsoleApp，引用了程序集 System.Runtime和System.Console。还有几个已配置的程序集属性，例如AssemblyCompanyNameAttribute、AssemblyConfigurationAttribute、AssemblyDescriptionAttribute、AssemblyFileVersionAttribute以及其他。

图 19-3

描述应用程序的程序集元数据可以使用 Visual Studio 配置；方法是在 Project Properties 中选择 Package (参见图 19-4)。

当然，还可以直接编辑项目文件(代码文件 ConsoleApp/ConsoleApp.csproj)：

```
<Project Sdk="Microsoft.NET.Sdk">
```

```
    <PropertyGroup>
      <OutputType>Exe</OutputType>
      <TargetFramework>netcoreapp2.0</TargetFramework>
      <Authors>Christian Nagel</Authors>
      <Company>CN innovation</Company>
      <Product>Sample App</Product>
      <Description>Sample App for Professional C#</Description>
      <Copyright>Copyright (c) CN innovation</Copyright>
      <PackageProjectUrl>https://github.com/ProfessionalCSharp
      </PackageProjectUrl>
      <RepositoryUrl>https://github.com/ProfessionalCSharp/ProfessionalCSharp7
      </RepositoryUrl>
      <RepositoryType>git</RepositoryType>
      <PackageTags>Wrox Press, Sample, Libraries</PackageTags>
    </PropertyGroup>

  </Project>
```

**注意：**

*在前面的项目中，这些元数据信息通常使用全局 C#属性添加到文件 AssemblyInfo.cs 中。仍然可以这样做，但是需要把 csproj 文件配置为不自动生成属性。*

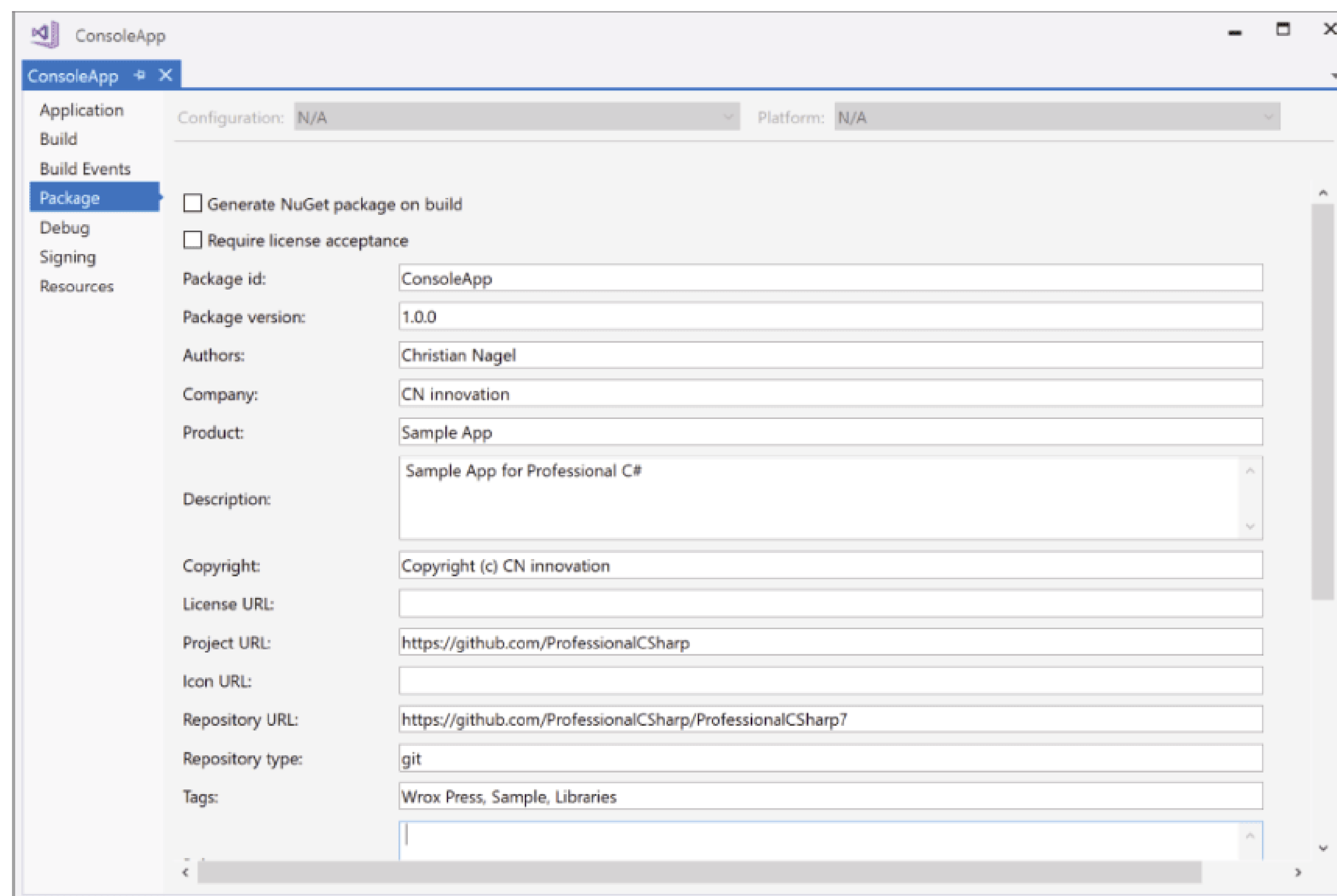



图 19-4

## 19.3 创建库

可以通过创建库来使用共享代码。在 Visual Studio 2017 中，有许多创建库的选项，如下所示：

- Class Library (.NET Core)
- Class Library (.NET Standard)
- Class Library (.NET Framework)
- WPF Custom Control Library (.NET Framework)
- WPF User Control Library (.NET Framework)
- Windows Forms Control Library (.NET Framework)
- Class Library (Universal Windows)
- Class Library (Legacy Portable)

- Shared Project

上面列出的 Shared Project 并不是一个真正的类库，但是可以使用它来共享多个项目中的代码。

在前面的列表中，使用.NET Framework 标识的类库应该与.NET Framework 一起使用，它们可以有特定的限制。WPF User Control Library 和 WPF Custom Control Library 只能在 WPF 应用程序中使用。类似地，Windows Forms Control Library 只能在 Windows Forms 应用程序中使用。

Class Library(Legacy Portable)已经在名字中包含 Legacy 了。这个项目模板最初称为 Portable Class Library (PCL)，不应该用于新的应用程序。这个库能够共享不同技术之间的代码，例如在 Silverlight、WPF、Xamarin、.NET Core 等之间共享代码。根据平台和版本的选择，可以使用不同的 API。平台越多，选择的版本越老，可用的 API 就越少。随着添加的平台越来越多，就增加了定义的复杂性，也增加了在可移植的库中使用可移植库的复杂性。

.NET 标准为可移植的库提供了替代品。

## 19.3.1　.NET 标准

.NET 标准对可用的 API 进行了线性定义，这与可用于可移植库的 API 的矩阵定义不同。.NET 标准的每个版本都添加了 API，而 API 从未删除。

.NET 标准的版本越高，可以使用的 API 越多。然而，.NET 标准并没有实现 API；它只是定义了需要由.NET 平台实现的 API。这可以与接口和具体类相比较。接口为需要由类实现的成员定义了协定。在.NET 标准中，.NET 标准指定了哪些 API 需要可用，以及需要实现这些 API 的.NET 平台——支持特定版本的标准。

在 https://github.com/dotnet/standard/tree/master/docs/versions 中可以找到哪些 API 可用于哪个标准版本，以及标准之间的差异。

.NET 标准的每个版本都将 API 添加到标准中：

- .NET Standard 1.1 在.NET Standard 1.0 中添加了 2414 个 API。
- 版本 1.2 只添加了 46 个 API。
- 在 1.3 版本中，添加了 3314 个 API。
- 1.4 版本只添加了 18 个加密 API。
- 1.5 版本主要增强了反射支持，增加了 242 个 API。
- 1.6 版本增加了更多的加密 API 和增强的正则表达式，共额外添加了 146 个 API。

在.NET Standard 2.0 中，微软进行了大量的投资，使其更容易将旧应用程序迁移到.NET Core。在这个新标准中，添加了 19507 个 API。并非所有这些 API 都是新的。有些已经在.NET Framework 4.6.1 中实现了。例如，像 DataSet、DataTable 之类的旧 API 现在可用于.NET 标准。这是为了便于将旧应用程序迁移到.NET 标准中。.NET Core 需要大量的投资，因为.NET Core 2.0 实现了.NET Standard 2.0。

哪些 API 不是标准的，永远都不会变成标准？特定于平台的 API 不可能成为.NET 标准的一部分。例如，Windows Presentation Foundation (WPF)和 Windows Forms 定义了特定于 Windows 的 API，而这些 API 不会成为标准。但是，可以创建 WPF 和 Windows Forms 应用程序，并在其中使用.NET 标准库。不能创建包含 WPF 或 Windows Forms 控件的.NET 标准库。

测试过的新 API 将首先进入.NET Core。一旦 API 稳定下来，就可以用于.NET 标准的未来版本。

下面讨论更多关于.NET 标准平台支持的细节。如果需要支持 Windows Phone Silverlight 8.1，就要将库限制为可用于.NET Standard 1.0 中的 API。当然，现在通常不需要支持任何 Silverlight 版本。下面介绍更重要的.NET 平台。

对于使用通用 Windows 平台的库，需要注意要支持的构建号。如果只支持 Fall Creators Update of Windows 10，就可以使用.NET Standard 2.0。为了支持 Creators Update 和旧的 Windows10 构建版本，可以升级到.NET Standard 1.4。在这种情况下，新版本是不可用的。如果创建一个包含 ASP.NET Core 1.1 控制器的库，库就需要是标准的 1.6 版。

**注意：**
要支持尽可能多的平台，需要选择较低的.NET 标准版本。要获得更多的 API，请选择更高.NET 标准版本。

### 19.3.2 创建.NET 标准库

要创建.NET 标准库，可以通过如下命令使用.NET Core CLI 工具。

```
dotnet new classlib
```

使用.NET Core 2.0 CLI工具，将创建一个具有这个csproj定义的库(代码文件UsingLibs/SimpleLib/SimpleLib.csproj)：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
  </PropertyGroup>
</Project>
```

通过更改 TargetFramework 元素，可以更改.NET 标准库的版本。在 Visual Studio 中，可以使用 Project Properties 的 Application Settings 设置来更改.NET 标准版本(参见图 19-5)。

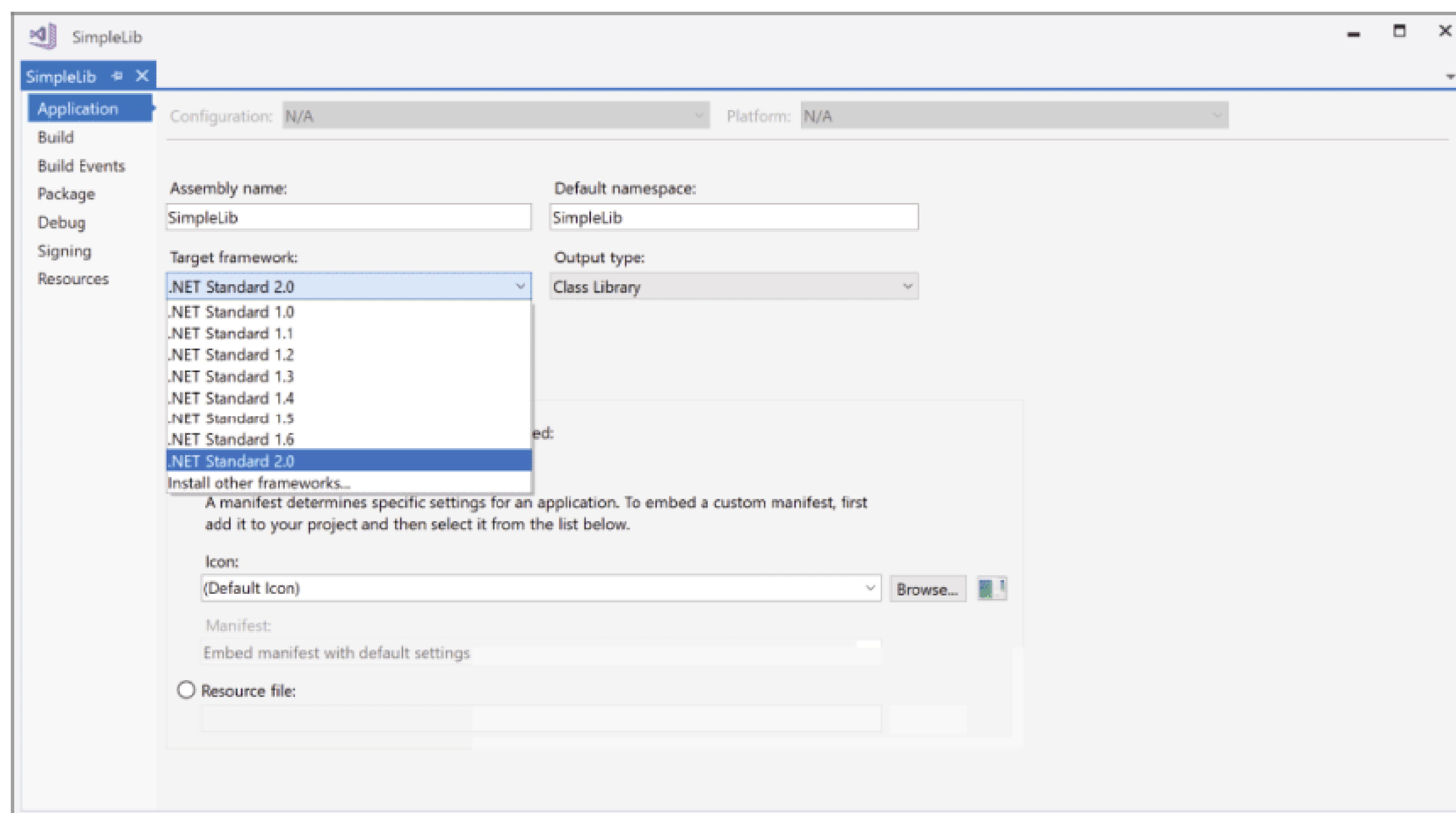



图 19-5

### 19.3.3 解决方案文件

使用多个项目(例如，一个控制台应用程序和一个库)时，使用解决方案文件是很有帮助的。在.NET Core CLI 工具的新版本中，可以在命令行中使用解决方案，也可以在 Visual Studio 中使用它们。例如，下面的命令

```
> dotnet new sln
```

在当前目录中创建一个解决方案文件。

使用 dotnet sln add 命令，可以向解决方案文件中添加项目：

```
dotnet sln add SimpleLib/SimpleLib.csproj
```

项目文件添加到解决方案文件中，如下面的代码片段所示(解决方案文件 UsingLibs\ UsingLibs.sln)：

```
Microsoft Visual Studio Solution File, Format Version 12.00
# Visual Studio 15
VisualStudioVersion = 15.0.26124.0
MinimumVisualStudioVersion = 15.0.26124.0
Project("{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}") = "SimpleLib",
  "SimpleLib\SimpleLib.csproj", "{C58F9225-7407-45A0-932A-81AC3906F228}"
EndProject
Project("{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}") = "ConsoleApp",
```

```
    "ConsoleApp\ConsoleApp.csproj", "{31E6F88A-C0BC-4277-A9E9-19DAFDEB1A7A}"
EndProject
Global
# ...
```

使用 Visual Studio 时，可以在 Solution Explorer 中选择解决方案，来添加新项目。从上下文菜单中选择 Add，然后选择 Existing Project 来添加现有项目。

## 19.3.4　引用项目

使用 dotnet add reference 命令可以引用一个库。当前目录只需要定位在应该添加库的项目的目录中：

```
dotnet add reference ..\SimpleLib\SimpleLib.csproj
```

在 csproj 文件中使用 ProjectReference 元素来添加引用(项目文件 UsingLibs/ConsoleApp/ConsoleApp.csproj)：

```
<Project Sdk="Microsoft.NET.Sdk">

  <ItemGroup>
    <ProjectReference Include="..\SimpleLib\SimpleLib.csproj" />
  </ItemGroup>

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

</Project>
```

在 Visual Studio 中使用 Solution Explorer，可以选择 Dependencies 节点，然后从 Project 菜单中选择 Add Reference 命令，从而向其他项目添加项目。打开的对话框如图 19-6 所示。

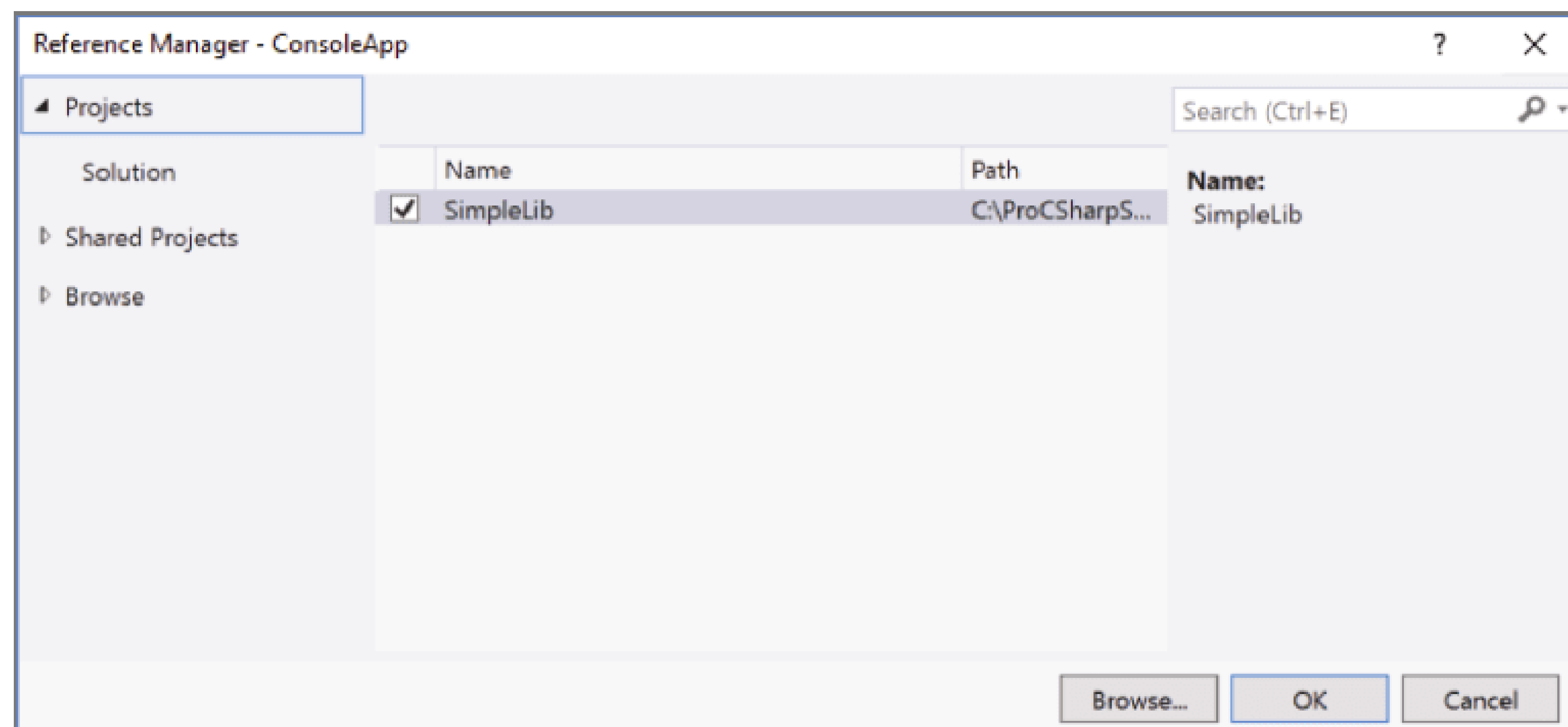



图 19-6

## 19.3.5　引用 NuGet 包

如果库已经打包在 NuGet 包中，则可以直接使用命令 dotnet add package 来引用 NuGet 包。

```
dotnet add package Microsoft.Composition
```

它没有像以前那样添加一个 ProjectReference，而是添加了一个 PackageReference。

```
<Project Sdk="Microsoft.NET.Sdk">
  <ItemGroup>
    <ProjectReference Include="..\SimpleLib\SimpleLib.csproj" />
  </ItemGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.Composition" Version="1.0.31" />
  </ItemGroup>
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>
</Project>
```

为了请求包的特殊版本，可以使用.NET Core CLI 命令指定-version 选项。在 Visual Studio 中，可以使用 NuGet 包管理器(参见图 19-7)找到包，并选择包的一个特定版本。使用此工具，还可以获得项目的细节、与项目的链接和许可信息。

**注意：**
在 www.nuget.org 上，并不是所有的包都对应用程序有用。应该检查许可信息，以确保许可证符合项目需求。另外，应该检查包的作者。如果它是一个开源的包，那么它背后的社区有多活跃？

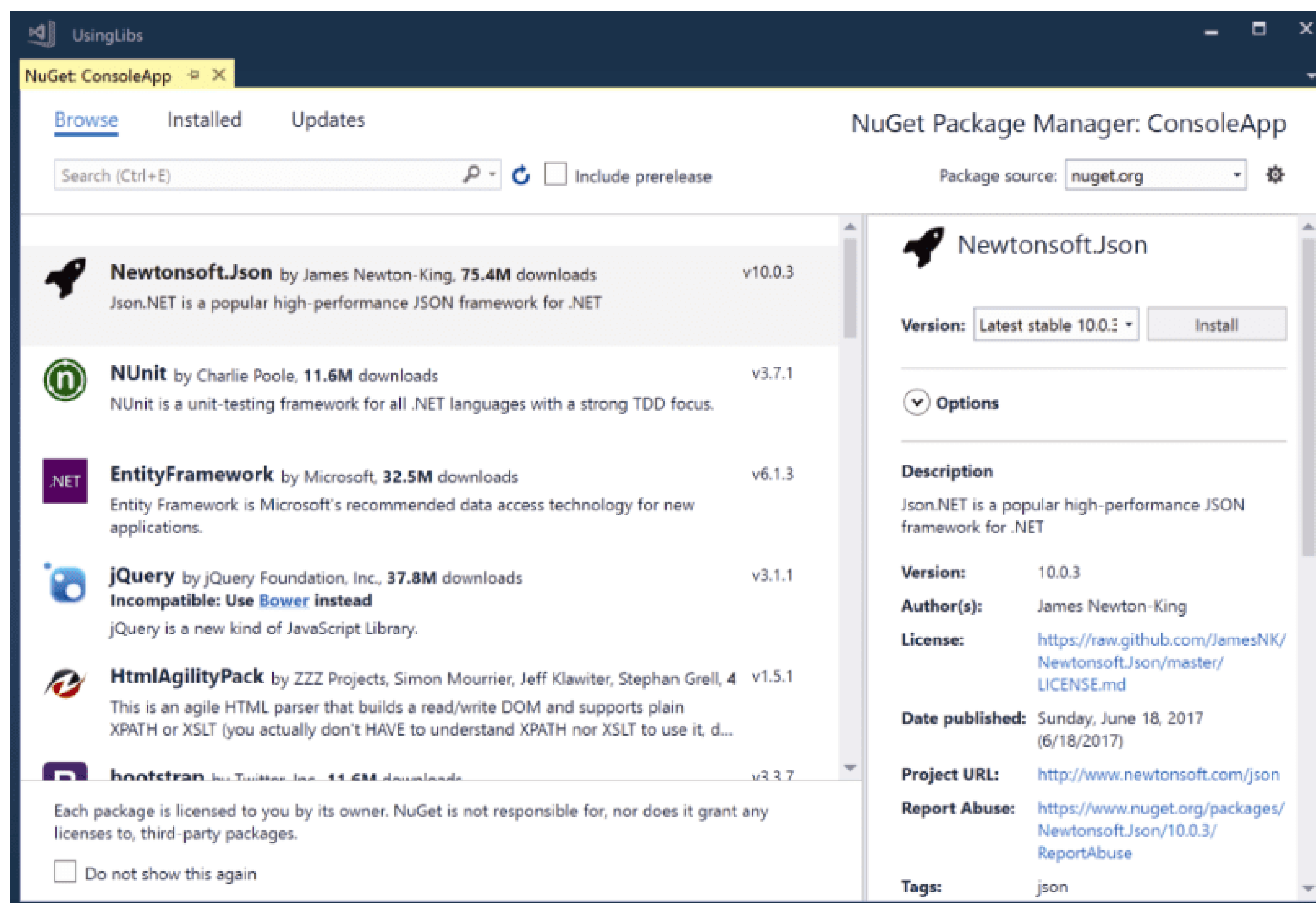



图 19-7

## 19.3.6 NuGet 的来源

包从哪里来？www.nuget.org 是微软和第三方上传.NET 包的服务器。首次从 NuGet 服务器上下载包之后，包就存储在用户配置文件中。因此，用相同的包创建另一个项目会快得多。

在 Windows 上，用户配置文件中包的目录是%userprofile%\.nuget\packages。也使用其他临时目录。要获取关于这些目录的所有信息，最好安装 NuGet 命令行实用程序，它可以从 https://dist.nuget.org/下载。

要查看全局包、HTTP 缓存和 temp 包的文件夹，可以使用 nuget local：

```
> nuget locals all -list
```

在一些公司中，只允许使用经过批准并存储在本地 NuGet 服务器中的包。NuGet 服务器的默认配置在% appdata % /nuget 目录的文件 NuGet.Config 中。

默认配置类似于下面的 NuGet.Config 文件。包从 https://api.nuget.org 和本地的 NuGetFallbackFolder 中加载。

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <packageSources>
    <add key="nuget.org" value="https://api.nuget.org/v3/index.json"
        protocolVersion="3" />
    <add key="CliFallbackFolder"
        value="C:\Users\chris\.dotnet\NuGetFallbackFolder" />
  </packageSources>
</configuration>
```

可以通过添加和删除包源来更改默认值。

微软并没有在主 NuGet 服务器上的日常构建中存储包。为了使用.NET Core NuGet 包的日常构建，需要配

置其他 NuGet 服务器。还可以配置本地目录，在其中可以放置定制的 NuGet 包。以下 NuGet.Config 文件添加了一个本地目录和.NET Core 包到包源的夜间提要。

```
<configuration>
  <packageSources>
    <add key="local packages" value="C:\git\mypackages" />
    <add key="dotnet-core"
      value="https://dotnet.myget.org/F/dotnet-core/api/v3/index.json" />
  </packageSources>
</configuration>
```

可以将配置文件放置到项目目录中，而不是更改默认配置文件。这样，配置的包源只对项目有效。

### 19.3.7　使用.NET Framework 库

如本章前面所述，.NET Standard 2.0 的一大优点是来自.NET Framework 的额外 API。这便于将旧.NET 应用程序迁移到新的.NET。

.NET 标准库可以在许多不同的.NET 技术中使用。当然，可以从.NET Core 中引用.NET 标准库，也可以从 Mono 和.NET Framework 项目中引用。WPF 应用程序可以使用.NET 标准库。使用.NET Core 1.0 创建的库已经可以实现这样的场景。但是现在，在.NET Standard 2.0 中，扩展了互操作场景。只要.NET Framework 库只使用可用于.NET 标准的 API，也可以从.NET 标准库中引用旧的.NET Framework 库。

这怎么可能？尝试使用早期.NET Core 版本的互操作场景常常会导致兼容性错误。例如，对象类型定义了两次。这些问题的原因很容易解释。使用老的.NET 技术，例如.NET Framework，在 mscorlib 中定义了 object 类等核心类型。使用诸如.NET Core 这样的新技术，对象类型在 System.Runtime 中定义。使用这两种方法，通常会得到对象和其他核心类型的副本。

.NET Standard 2.0 改变了这种行为。.NET 标准定义了一个 API 集，而不是一个实现。API 的完整实现需要在.NET 平台(如.NET Framework 和.NET Core)中完成。标准是使用类型转发实现的，它将标准的类型转发到具体的实现中。列出.NET 标准中类型的新库是 NetStandard.dll。这个库对每个平台都不一样。NetStandard.dll 列出类型，但不包含任何实现。相反，这个库包含了特定实现的类型转发器。例如，在.NET Framework 项目中添加.NET 标准库时， NetStandard.dll 会自动引用，并包含从 System.Console 类到 mscorlib 程序集的一个类型转发器：

```
.class extern forwarder System.Console
{
  .assembly extern mscorlib
}
```

因此，对于.NET Core 项目，NetStandard.dll 包含从 System.Console 到库 System.Console 的重定向。

```
.class extern forwarder System.Console
{
  .assembly extern System.Console
}
```

下面使用类 Legacy 创建一个旧.NET Framework 库来实现这一点。方法 ConsoleMessage 和 WindowsMessage 只写入输出。ConsoleMessage 把输出写进控制台。WindowsMessage 利用.NET Framework 库 System.Windows.Forms，并打开一个消息框。另外，ShowConsoleType 方法提供了 Console 类来源的信息(代码文件 UsingLegacyLibs/DotnetFrameworkLib/Legacy.cs)：

```
public class Legacy
{
  public static void ConsoleMessage(string message)
  {
    Console.WriteLine($"From the .NET Framework Lib: {message}");
  }

  public static void ShowConsoleType()
  {
    Console.WriteLine($"The type {nameof(Console)} is from " +
      $"{Assembly.GetAssembly(typeof(Console)).FullName}");
  }
```

```
    public static void WindowsMessage(string message)
    {
      MessageBox.Show($"Windows Forms: {message}");
    }
  }
```

在这个场景中需要注意的是，Console 类是.NET 标准的一部分，但是 MessageBox 类不是。Windows Forms 是特定于 Windows 的，不会成为.NET 标准的一部分。

接下来，.NET 标准库引用这个.NET Framework 库。通过添加对项目的引用，可以以简单的方式处理它。项目文件包含一个对.NET Framework 库的 ProjectReference(项目文件 UsingLegacyLibs/DotnetStandardLib/DotnetStandardLib.csproj)：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <ProjectReference Include="..\DotnetFrameworkLib\DotnetFrameworkLib.csproj" />
  </ItemGroup>
</Project>
```

在.NET 标准库中定义的 Wrapper 类只是将方法调用转发到.NET Framework 库。当引用.NET Framework 库时，构建.NET 标准库就没有编译错误(代码文件 UsingLegacyLibs/DotnetStandardLib/Wrapper.cs)：

```
public class Wrapper
{
  public static void ConsoleMessage(string message) =>
    Legacy.ConsoleMessage(message);

  public static void WindowsMessage(string message) =>
    Legacy.WindowsMessage(message);

  public static void ShowConsoleType() => Legacy.ShowConsoleType();
}
```

接下来，.NET Core 控制台应用程序使用了 Wrapper 类。Program 类调用三个方法。但是，当调用 WindowsMessage 方法时，检查 FileNotFoundException 异常的处理(代码文件 UsingLegacyLibs/UsingLegacyLibs/Program.cs)：

```
static void Main()
{
  Wrapper.ConsoleMessage("Hello from .NET Core");
  Wrapper.ShowConsoleType();
  try
  {
    Wrapper.WindowsMessage("Hello from .NET Core");
  }
  catch (FileNotFoundException ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

应用程序的运行结果如下所示。Console 类来自 System.Console 程序集。调用 WindowsMessage 会导致一个 FileNotFoundException，因为找不到 System.Windows Forms 程序集：

```
From the .NET Framework Lib: Hello from .NET Core
The type Console is from System.Console, Version=4.1.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a
Could not load file or assembly 'System.Windows.Forms, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089'.
The system cannot find the file specified.
```

在创建.NET Framework 控制台应用程序时，不需要将 WindowsMessage 方法的调用封装到 try/catch 处理程序中，因为 System.Windows.Forms 是可用的(代码文件 UsingLegacyLibs/DotnetFrameworkApp/Program.cs)：

```
static void Main()
{
  Wrapper.ConsoleMessage("Hello from the .NET Framework");
  Wrapper.ShowConsoleType();
  Wrapper.WindowsMessage("Hello from the .NET Framework");
}
```

运行此应用程序会表明，Console 类来自于 mscorlib 程序集并为 MessageBox 弹出一个窗口：

```
From the .NET Framework Lib: Hello from the .NET Framework
The type Console is from mscorlib, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089
```

**警告：**

很容易从.NET Core应用程序中引用.NET Framework程序集，并构建应用程序。但是，这并不意味着.NET Framework程序集的每个功能都是可用的——只有.NET标准中定义的类型可用。使用这些类型就可以在Linux上使用.NET Framework库。

通常，如果类型不可用，则最好选择编译错误。把.NET Framework 库重建为.NET 标准库，就提供了这个特性。如果没有可用的源代码，比如来自还未提供.NET 标准库的供应商，仍然可以使用.NET Core 中的库。要检查二进制文件，并确定哪些类型与哪个.NET 标准版本不兼容，可以使用.NET 可移植性分析器(.NET Portability Analyzer)，它可用作命令行工具和 Visual Studio 扩展(参见 https://github.com/microsoft/dotnet -apiport)。

在示例.NET Framework 库中运行.NET 可移植性分析器，展示了关于支持特定类型和成员的平台版本的信息(见表 19-1)。

表 19-1 支持特定类型和成员的平台版本的信息

| 目标类型 | 目标成员 | .NET Core | .NET Framework | .NET 标准 |
| --- | --- | --- | --- | --- |
| DialogResult | DialogResult | 不支持 | 支持: 1.1+ | 不支持 |
| MessageBox | MessageBox | 不支持 | 支持: 1.1+ | 不支持 |
| MessagBox | Show | 不支持 | 支持: 1.1+ | 不支持 |
| Assembly | GetAssembly | 支持: 2.0+ | 支持: 1.1+ | 支持: 2.0+ |

## 19.4 使用共享项目

共享项目并不是真正的库，但它们仍然有助于共享代码。共享项目可以替代一个库来共享代码，但是包含代码和引用共享项目的项目。通过这种方式，可以将特定于平台的代码添加到共享项目中。然而，这个特性只有在没有太多代码差异的情况下才有用。当存在大量的代码差异时，创建特定于平台的库可能更好。

通过下面的代码示例，创建了.NET Core 应用程序和通用 Windows 应用程序，它们都引用了共享项目。共享项目包含可以同时用于两个平台的代码，但每个都包含特定于平台的代码。

不同之处在于不能用于所有地方的名称空间。可以使用预处理器指令来检查条件编译符号。预处理器指令 Windows_UWP 是用通用 Windows 应用程序定义的(代码文件 UsingASharedProject/SharedProject/Message.cs)：

```
using System;
using System.Threading.Tasks;
#if WINDOWS_UWP
using Windows.UI.Popups;
#endif
```

Message 类定义 Show、ShowAsync 和 Add 方法。Add 方法适用于每个平台。只有在定义 NETCOREAPP2_0 指令时，Show 方法才可用，而 ShowAsync 方法仅适用于 UWP 应用程序。类可以用 internal 访问修饰符定义，因为它不在程序集之外使用：

```
internal class Message
{
#if NETCOREAPP2_0
  public static void Show(string message)
  {
    Console.WriteLine(message);
  }
```

```
#elif WINDOWS_UWP
  public static async Task ShowAsync(string message)
  {
    await new MessageDialog(message).ShowAsync();
  }
#endif
  public static int Add(int x, int y) => x + y;
}
```

使用 Visual Studio，可以从 Reference Manager 选择 Shared Project，以添加共享项目，如图 19-8 所示。其中包含了源代码，Import 元素与项目文件一起使用(项目文件 UsingASharedProject/UsingASharedProject/UsingASharedProject.csproj)：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>
  <Import Project="..\SharedProject\SharedProject.projitems" Label="Shared" />
</Project>
```

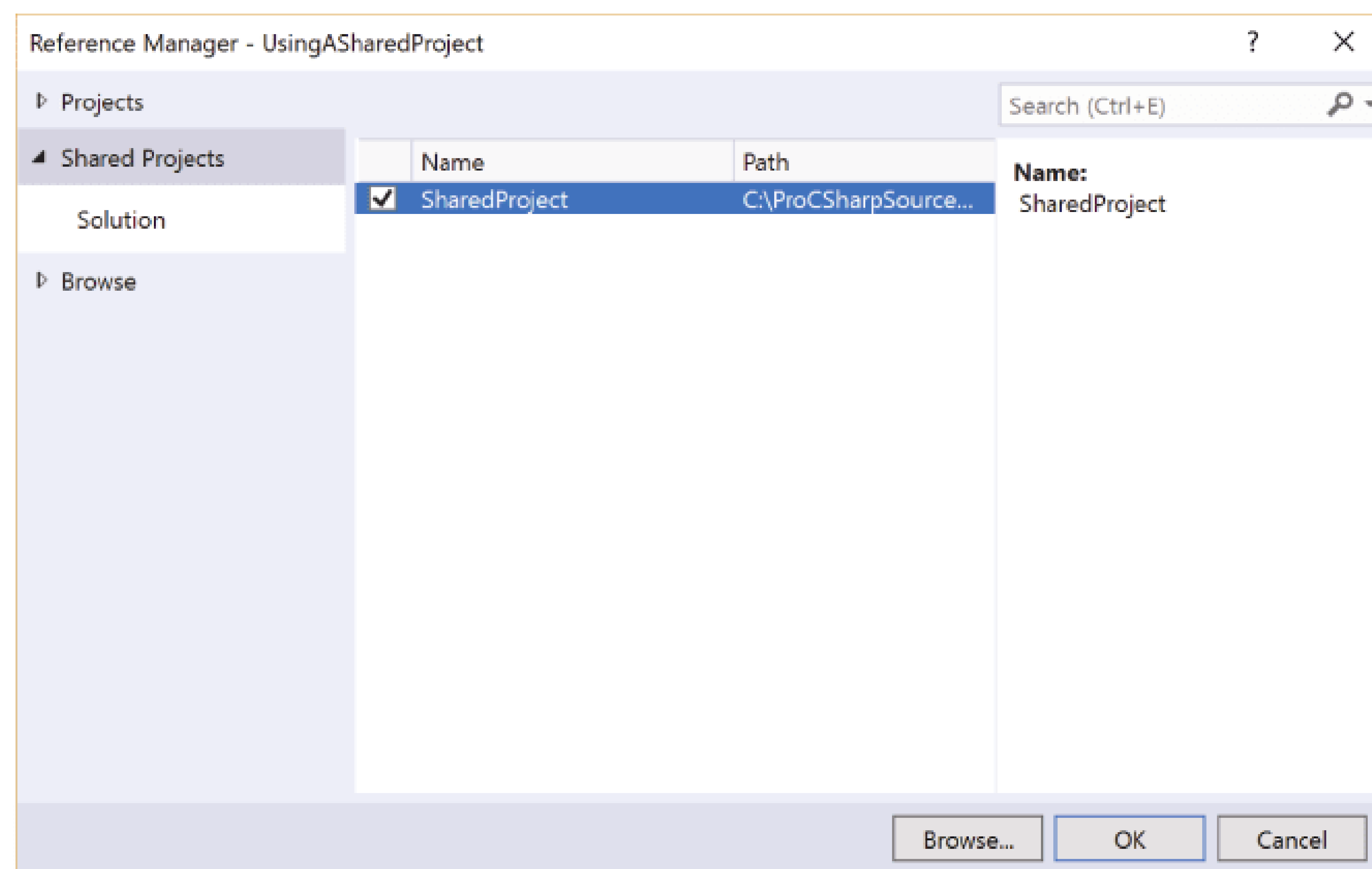



图 19-8

现在，Message 类的用法可以与同一个项目的类相似(代码文件 UsingASharedProject/UsingASharedProject/Program.cs)：

```
using SharedProject;

namespace UsingASharedProject
{
  class Program
  {
    static void Main()
    {
      Message.Show(".NET Core");
    }
  }
}
```

在 UWP 应用程序中，Message 类是用法相同。这里，在按钮的单击处理程序中调用 ShowAsync 方法(代码文件 UsingASharedProject/UniversalApp/MainPage. xaml.cs)：

```
private async void OnButtonClick(object sender, RoutedEventArgs e)
{
  await Message.ShowAsync("Hello from UWP");
}
```

**注意：**
在共享项目的源代码中使用 Visual Studio 编辑器时，可以在编辑器顶部选择下拉视图，以选择要处理的当前项目。这会基于已定义的预处理器定义灰显目前不可用的代码。

# 19.5　创建 NuGet 包

前面介绍了共享项目，现在继续讨论库——从库中创建 NuGet 包。使用.NET Core CLI 工具和 Visual Studio 2017，可以轻松创建 NuGet 包。

## 19.5.1　NuGet 包和命令行

关于 NuGet 包的元数据信息可以添加到项目文件 csproj 中。要从命令行上创建 NuGet 包，可以使用 dotnet pack 命令：

```
> dotnet pack --configuration Release
```

记住要设置配置。默认情况下会构建 Debug 配置。成功打包后，NuGet 包就放在目录 bin/Release 或相关目录中，这取决于所选择的扩展名为.nupkg 的配置文件。.nupkg 文件是一个 zip 文件，包含带有附加元数据的二进制文件。可以将该文件重命名为 zip 文件，以查看其内容。

以将生成的 NuGet 包复制到系统的文件夹或网络共享中，使其可用于团队。样本复制到文件夹 c:/mypackage 中。要使用这个文件夹，NuGet.config 文件可以改为包含这个包源。也可以使用 dotnet add package 命令直接引用文件夹：

```
> dotnet add package SampleLib --source c:/MyPackages
```

## 19.5.2　支持多个平台

.NET Standard 2.0 已经增强为支持更多的 API，它们在每个新的.NET 平台上都可用。但是，对于某些应用程序来说，这仍然不够。而是可能需要通过旧库来使用.NET 标准中没有的多个 API，或者可能需要 API 之间不同的特性(如 WPF、UWP 和 Xamarin)。.NET Core 提供的特性也比.NET 标准提供的功能多。

之前探讨了如何使用共享项目来支持不同的平台。另一种方法是使用相同的源代码创建不同的二进制文件，如下一个示例所示。

示例库 SampleLib 通过不同的二进制文件支持.NET Standard 2.0 和.NET Framework 4.7。为了构建多个二进制文件，可以将 TargetFramework 元素更改为 TargetFrameworks，并将目标框架中应创建二进制文件的所有目标框架别名(target framework monikers)都列在其中。这个例子添加了目标框架别名 netstandard2.0 和 net47。对于基于目标框架的代码差异，定义了不同的条件编译符号(项目文件 CreateNuGet/SampleLib/SampleLib.csproj):

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFrameworks>netstandard2.0;net47</TargetFrameworks>
    <!-- Metadata information -->
  </PropertyGroup>

  <PropertyGroup Condition="'$(TargetFramework)'=='netstandard2.0'">
    <DefineConstants>NETSTANDARD2_0</DefineConstants>
  </PropertyGroup>

  <PropertyGroup Condition="'$(TargetFramework)'=='net47'">
    <DefineConstants>DOTNET47</DefineConstants>
  </PropertyGroup>
</Project>
```

**注意：**
目标框架别名的列表显示在 https://docs.microsoft.com/nuget/schema/target-frameworks 上。

在代码中，可以很容易地看到不同的功能，不同的字符串用不同的值初始化，这个值从 Show 方法返回(代码文件 CreateNuGet/SampleLib/Demo.cs)：

```
public class Demo
{
#if NETSTANDARD2_0
  private static string s_info = ".NET Standard 2.0";
```

```
#elif DOTNET47
  private static string s_info = ".NET 4.7";
#else
  private static string s_info = "Unknown";
#endif

  public static string Show() => s_info;
}
```

通过这种设置，可以使用多个目标框架构建应用程序，并为每个目标框架创建一个 DLL。创建 NuGet 时，会创建一个包含所有库的包。

创建.NET Core 控制台应用程序时，也可以为多个目标框架构建应用程序。与之前的库一样，使用控制台应用程序配置多个目标框架。控制台应用程序将针对.NET Core 2.0 和.NET Framework 4.7 构建(项目文件 CreateNuGet/DotnetCaller/DotnetCaller.csproj)：

```
<TargetFrameworks>netcoreapp2.0;net47</TargetFrameworks>
```

还可以为特定的目标框架添加包。为此，可以添加--framework 选项，并使用 dotnet add package 命令：

```
> dotnet add package SampleLib --framework net47 --source c:/MyPackages
```

这显示了基于项目文件中目标框架的条件，如下所示：

```
<ItemGroup Condition="'$(TargetFramework)' == 'net47'">
  <PackageReference Include="SampleLib" Version="1.2.0" />
</ItemGroup>
```

对于样例应用程序，需要相同的包，但是需要选择包中的不同程序集。这是根据项目自动完成的，而包只需要添加到项目中。这里显示了控制台应用程序的完整项目文件(项目文件 CreateNuGet/DotnetCaller/DotnetCaller.csproj)：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFrameworks>netcoreapp2.0;net47</TargetFrameworks>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="SampleLib" Version="1.2.0" />
  </ItemGroup>
</Project>
```

构建控制台应用程序，会创建多个二进制文件，其中包含对不同库的引用。

```
> dotnet run --framework net47
```

输出如下：

```
.NET 4.7
```

使用.NET Core 变体：

```
> dotnet run --framework netcoreapp2.0
```

输出如下：

```
.NET Standard 2.0
```

请注意，选择不同的.NET Framework 版本，如.NET 4.6.1，也会得到.NET Standard 2.0。这是因为使用.NET Framework 4.7 构建的库与.NET 4.6.1 不兼容(只有较新版本的.NET Framework 兼容)，但是.NET 4.6.1 与.NET Standard 2.0 兼容，所以这个库匹配。

### 19.5.3 NuGet 包与 Visual Studio

Visual Studio 2017 允许创建包。在 Solution Explorer 中选择项目时，可以打开上下文菜单并选择 Pack，来创建 NuGet 包。在 Package 设置的项目属性中，还可以选择在每个构建上创建一个 NuGet 包。如果不打算在每个构建上分发包，那么这可能是多余的。但是，对于这个设置，应该配置包元数据、程序集和包版本(参见图 19-9)。

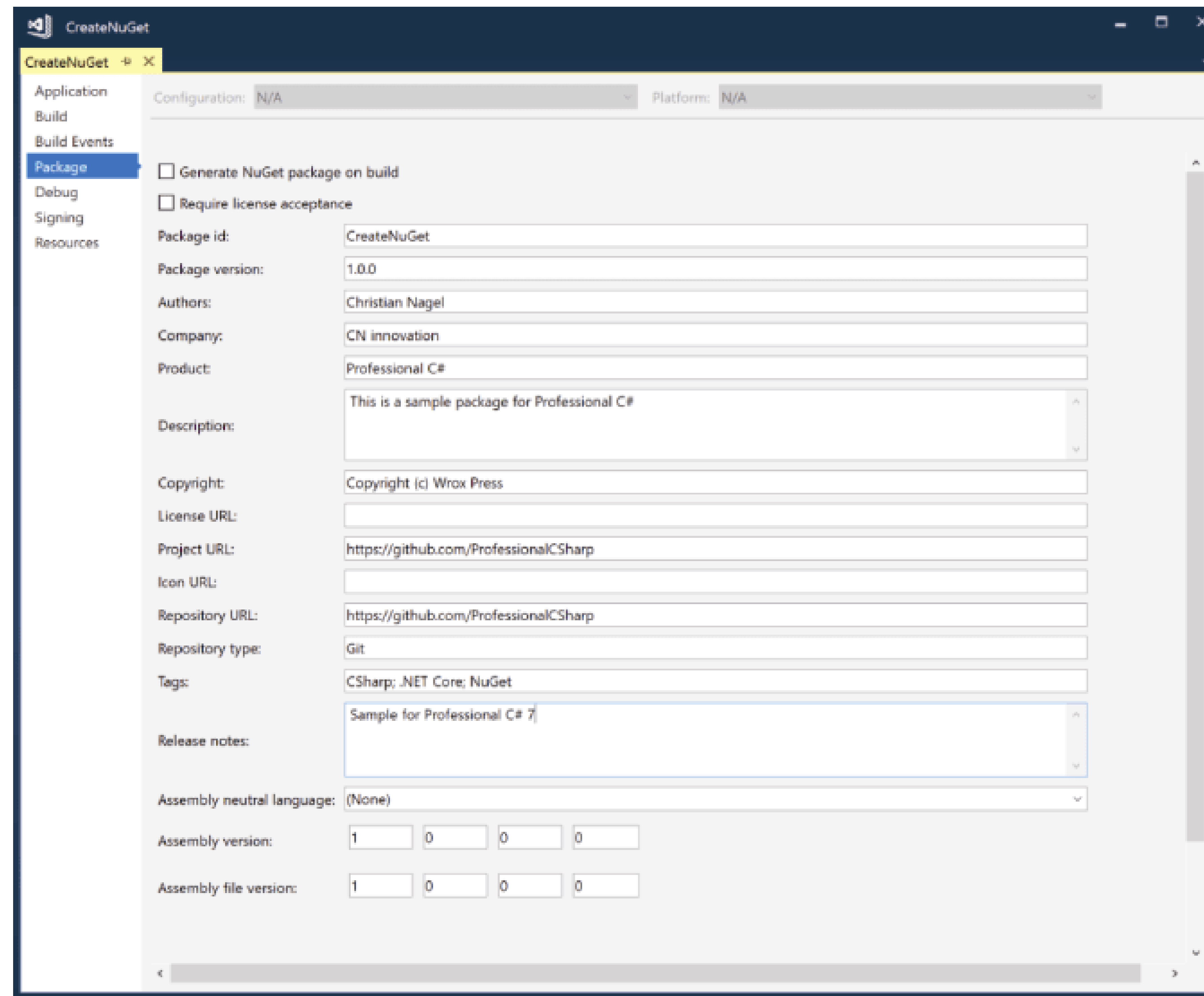



图 19-9

要在 Visual Studio 中使用包，可以在 Solution Explorer 中选择 Dependencies，打开上下文菜单，选择 Manage NuGet Packages。这将打开 NuGet Package Manager (参见图 19-10)，可以在其中选择包源(如果通过单击 Settings 图标来配置该包，则可以从本地文件夹中选择包)。可以浏览可用的包，查看安装在项目中的包，并检查包的更新是否可用。

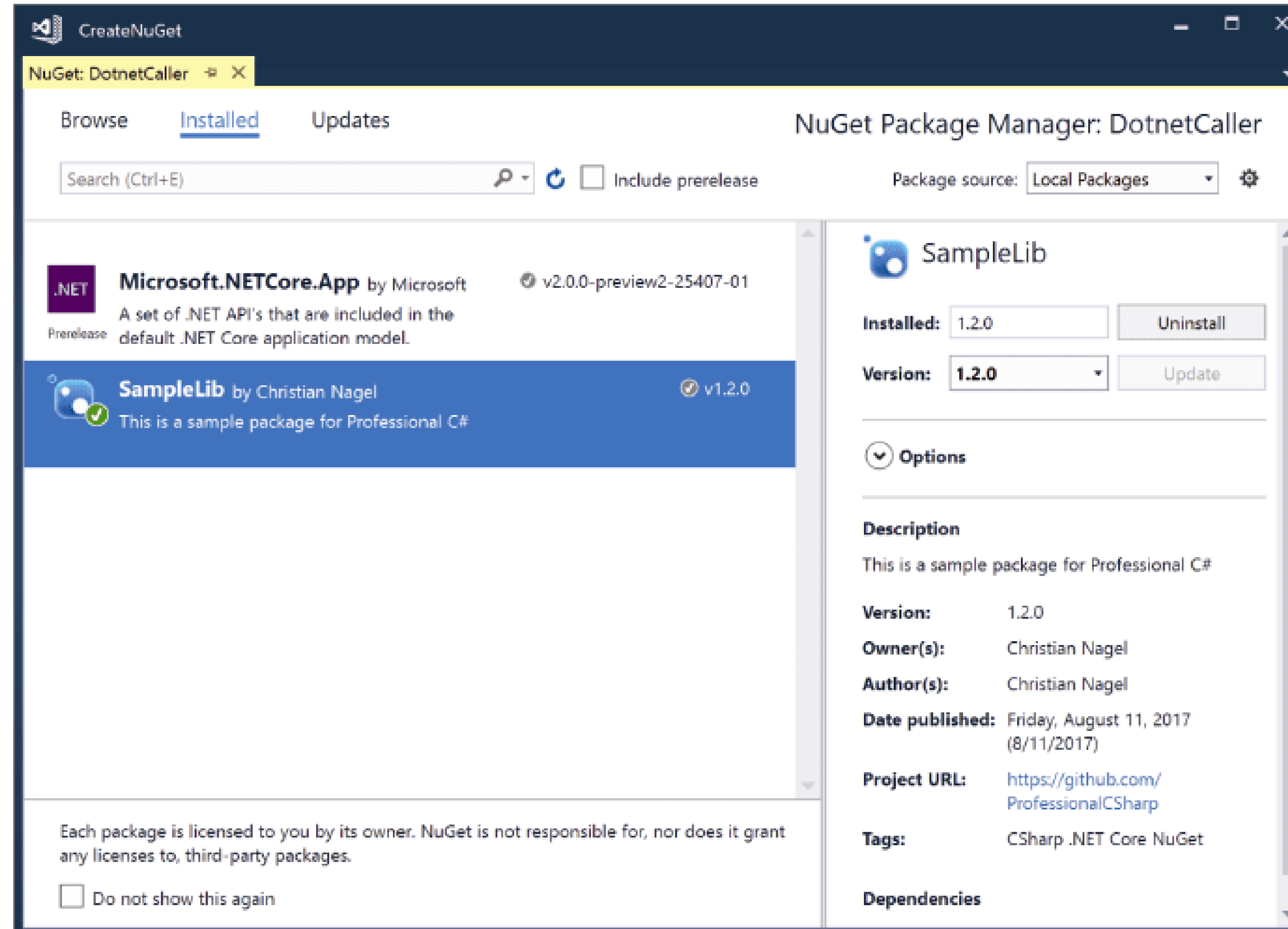



图 19-10

**注意：**

如果要创建 NuGet 包，不仅用于内部，还需要在 NuGet 服务器上发布，仅将库添加到 NuGet 文件中是不够的。还可以添加一个 readme 文件，添加相关项目的库，定义依赖项，并创建构建脚本。为此，需要从 www.nuget.org 中下载 NuGet 实用程序。访问 https://docs.microsoft.com/nuget 获取更多关于如何操作的信息。

## 19.6 小结

本章解释了 DLL、程序集和 NuGet 包之间的区别，了解了如何使用 NuGet 包创建和分发库。

.NET 标准定义了一个在不同的.NET 平台上实现的 API 集。讨论了.NET 标准库如何在.NET Core 和.NET Framework 中使用，以及类型如何来自不同的库，如 mscorlib 和 System.Runtime。

下一章将详细讨论一个重要的模式：依赖注入。该章将介绍另一种在不同平台上共享代码的方法：注入特定于平台的特性。

# 第20章 依赖注入

**本章要点**

- 定义依赖注入
- 使用依赖注入和 Microsoft.Extensions.DependencyInjection
- 处理服务的生命周期
- 使用选项和配置来初始化服务
- 使用 DI 为 WPF、UWP 和 Xamarin 创建平台独立性
- 使用其他依赖注入容器

**本章源代码下载：**

打开 www.wrox.com 的 Download Code 选项卡可下载本章源代码。源代码也可以在 DependencyInjection 目录的 https://github.com/ProfessionalCSharp/ProfessionalCSharp7 中找到。

本章代码分为以下几个主要的示例文件：

- NoDI
- WithDI
- WithDIContainer
- ServicesLifetime
- DIWithOptions
- DIWithConfigurations
- PlatformIndependenceSample
- DIWithAutofac

## 20.1 依赖注入的概念

更快的开发周期需要单元测试和更好的可更新性。更改一些代码，不应该导致意外位置出现错误。创建更模块化的、减少依赖项的应用程序，有助于防止这种错误。

依赖注入(Dependency Injection，DI)允许从类的外部注入依赖项，因此注入依赖项的类只需要知道一个协定(通常是 C#接口)。这个类可以独立于其对象的创建。

依赖注入更便于进行单元测试。在单元测试中，只需要测试特定的类，需要的依赖项可以替换为包含测试数据的特殊模拟类。

还可以使用不同的实现区分生产模式和开发模式。例如，在生产过程中，可能需要访问 SAP 服务器，或者可能需要对所有开发人员都无法访问的特定活动目录进行身份验证。在开发的每个调试会话期间，都不希望等待成功的身份验证，也不需要 SAP 服务器开发用户界面。在这里，可以给相同的协定使用不同的实现来模拟身份验证，可以使用测试数据而不是访问 SAP 服务器。

也可以在不同的平台上使用不同的实现。例如，可以创建一个.NET 标准库，在其中为 UWP、WPF 和 Xamarin 应用程序实现所有公共功能，并可以根据需要重定向到特定于平台的代码。

依赖注入还允许用自定义特性替换标准功能。ASP.NET Core 和 Entity Framework Core 主要基于依赖注入。这些技术使用数百个协定——例如，来找到控制器，将 HTTP 请求映射到控制器，将接收到的数据转换为参数，将数据库表映射到实体类型等。使用不同的实现，可以轻松地替换自定义功能。

DI 是敏捷软件开发和持续软件交付实践的核心模式。

依赖注入不需要依赖注入容器，但该容器有助于管理依赖项。依赖注入容器管理的服务列表越来越长，就可以看到它的优点。ASP.NET Core 和 Entity Framework Core 使用 Microsoft.Extensions.DependencyInjection 作为容器来管理所有依赖项，以此管理数百个服务。

尽管依赖注入和依赖注入容器在非常小的应用程序中会增加复杂性，但是一旦应用程序变得更大，需要多个服务，依赖注入就会降低复杂性，并促进非紧密绑定的实现。

本章从一个不使用依赖注入的小应用程序开始；在随后的示例中，它转换为使用依赖注入并使用依赖注入容器的应用程序。本章还介绍了服务的生命周期管理和配置。本章的最后一节将讨论如何使用依赖注入来覆盖与 WPF、UWP 和 Xamarin 相关的、特定于平台的服务。最后，本章讨论了第三方容器与 Microsoft.Extensions.DependencyInjection 的集成。

### 20.1.1 使用没有依赖注入的服务

下面的示例没有使用依赖注入；稍后将更改它，以使用依赖注入。所用的服务实现在类 GreetingService 中定义。这个类定义了返回字符串的 Greet 方法(代码文件 NoDI/GreetingService.cs)：

```
public class GreetingService
{
  public string Greet(string name) => $"Hello, {name}";
}
```

类 HomeController 使用这个服务。在 Hello 方法中，实例化了 GreetingService，并且调用 Greet 方法(代码文件 NoDI/HomeController.cs)：

```
public class HomeController
{
  public string Hello(string name)
  {
    var service = new GreetingService();
    return service.Greet(name);
  }
}
```

下面看看 Program 类的 Main()方法。其中实例化了 HomeController，调用 Hello 方法，将结果写入控制台(代码文件 NoDI/Program.cs)：

```
static void Main()
{
  var controller = new HomeController();
  string result = controller.Hello("Stephanie");
  Console.WriteLine(result);
}
```

程序运行时，把 Hello，Stephanie 写入控制台。这有什么问题吗？

HomeController 和 GreetingService 是紧密耦合的。要用不同的实现取代 HomeController 中的 GreetingService 并不容易。这个 GreetingService 是一个返回字符串的简单服务。在正常的应用程序中，场景通常更复杂——例

如，GreetingService 可能使用 HTTP 请求访问 API 服务，或者使用 Entity Framework 访问数据库。可能要更改在一个地方使用的服务，而不是查找使用服务的所有位置。

另外，为 HomeController 创建单元测试时，也会测试 GreetingService。在单元测试中，希望仅测试单个类的方法的功能，而不需要使用其他依赖项。在 HomeController 中，不能很容易地为单元测试替换 GreetingService。从技术上讲，为单元测试替换 GreetingService 方法的内部实现是可能的。使用 Microsoft Fakes 框架，可以通过替换 GreetingService 类的特定方法和属性，来更改方法的实现。这个变更是在单元测试中定义的，并且只有在单元测试运行时才会发生：通过另一个方法来“伪造”原来的方法。其实这有更好的方法：使用依赖注入。

下一节将介绍如何更改此实现，以使用依赖注入。

**注意：**

*Entity Framework Core 在第 26 章中详细介绍。第 28 章讨论了更多关于单元测试的信息。API 服务在第 32 章解释。*

## 20.1.2　使用依赖注入

下面使 HomeController 独立于 GreetingService 的实现。为此，可以创建接口 IGreetingService，它定义了 HomeController 所需的功能(代码文件 WithDI/IGreetingService.cs)：

```
public interface IGreetingService
{
  string Greet(string name);
}
```

GreetingService 现在实现了接口 IGreetingService(代码文件 WithDI/GreetingService.cs)：

```
public class GreetingService : IGreetingService
{
  public string Greet(string name) => $"Hello, {name}";
}
```

HomeController 现在只需要对一个对象的引用，该对象实现了接口 IGreetingService。它用 HomeController 的构造函数注入，分配给私有字段，通过方法 Hello 来使用(代码文件 WithDI/HomeController.cs)：

```
public class HomeController
{
  private readonly IGreetingService _greetingService;
  public HomeController(IGreetingService greetingService)
  {
    _greetingService = greetingService ??
      throw new ArgumentNullException(nameof(greetingService));
  }

  public string Hello(string name) => _greetingService.Greet(name);
}
```

在这个实现中，HomeController 利用了控制反转的设计原理。HomeController 没有像以前那样实例化 GreetingService。相反，定义由 HomeController 使用的具体类的控件在外部给出；换句话说，控制是反转的。

**注意：**

*控制反转也被称为好莱坞原则:不要给我们打电话；我们会给你打电话。*

*控制反转也减少了对不同技术的依赖，创建出更通用的代码。例如，可以在.NET 标准库中为 WPF、UWP 和 Xamarin 应用程序一同使用相同的视图模型和服务协定。对于 WPF、UWP 和 Xamarin，有些服务需要不同的实现。这种服务的实现可以来自于托管应用程序，而协定是在.NET 标准库中定义和使用的。阅读第 34 章可以获得关于视图模型的更多信息。*

类 HomeController 并没有依赖 IGreetingService 接口的具体实现。HomeController 可以使用实现了接口 IGreetingService 的任何类。这个类只需要实现这个接口的所有成员。现在，需要从外部注入依赖项，将具体的实现传递给 HelloController 类的构造函数。在样例代码中，使用构造函数注入的依赖注入模式实现了控制反转

的设计原则。它称为构造函数注入，因为接口是在构造函数中注入的。需要注入依赖项来创建一个 HomeController 实例。

下面修改 Main()方法，将 IGreetingService 的具体实现传递给 HomeController。这里，注入了依赖项(代码文件 WithDI/Program.cs)：

```
static void Main()
{
  var controller = new HomeController(new GreetingService());
  string result = controller.Hello("Matthias");
  Console.WriteLine(result);
}
```

为 HomeController 的 Hello 方法创建一个单元测试，可以注入不同的实现——例如，执行 IGreetingService 的 MockGreetingService。

示例应用程序目前非常小。唯一需要注入的是一个实现协定的具体类。这个类在实例化的同时实例化了 HomeController。在实际应用程序中，需要处理许多接口和实现，还需要共享实例。这样做的一个简单方法是使用依赖注入容器来管理所有依赖项。在下一节中，应用程序改为使用 Microsoft.Extensions.DependencyInjection 容器。

## 20.2 使用.NET Core DI 容器

在依赖注入容器中，可以在应用程序中有一个位置，在其中定义什么协定映射到哪个特定的实现上，还可以指定是应该将服务作为一个单例来使用，还是应该在每次使用时创建一个新实例。

在下一个示例中，使用前面创建的 GreetingService 来实现 IGreetingService 和 HomeController 类，但这次我们使用依赖注入容器。

示例 WithDIContainer 使用了如下 NuGet 包和名称空间：

**包**

Microsoft.Extensions.DependencyInjection

**名称空间**

System

Microsoft.Extensions.DependencyInjection

在 Program 类中，现在定义 RegisterServices 方法。在这里，实例化一个新的 ServiceCollection 对象。在添加了 NuGet 包 Microsoft.Extensions.DependencyInjection 之后，ServiceCollection 就在名称空间 Microsoft.Extensions.DependencyInjection 中定义。使用 AddSingleton 和 AddTransient 时，ServiceCollection 的扩展方法用来注册 DI 容器需要知道的类型。在示例应用程序中，GreetingService 和 HomeController 都在容器中注册，这允许从容器中检索 HomeController。

当请求 IGreetingService 接口时，会实例化 GreetingService 类。HomeController 本身没有实现接口。通过这个 DI 容器配置，当请求 HomeController 时，会实例化 HomeController。DI 容器配置还定义了服务的生命周期。对于 GreetingService，请求 IGreetingService 时总是返回相同的实例。这和 HomeController 是不同的。对于 HomeController，每次请求检索 HomeController 时，都会创建一个新的实例。使用 AddSingleton 和 AddTransient 方法指定服务的生命周期信息。本章后面将介绍这些服务的生命周期。

调用方法 BuildServiceProvider 会返回一个 ServiceProvider 对象，该对象可以用来访问已注册的服务(代码文件 WithDIContainer/Program.cs)：

```
static ServiceProvider RegisterServices()
{
  var services = new ServiceCollection();
  services.AddSingleton<IGreetingService, GreetingService>();
  services.AddTransient<HomeController>();
  return services.BuildServiceProvider();
}
```

**注意：**

在.NET Core 1.1 中，BuildServiceProvider 返回接口 IServiceProvider，而不是返回具体的类。在.NET Core 1.1 中，ServiceProvider 声明为 internal，因此只能通过它的公共接口 IServiceProvider 从外部使用。在.NET Core 2.0 中修改了实现方式，使 ServiceProvider 类变成 public，并更改 BuildServiceProvider 的方法声明，以返回 ServiceProvider。这允许直接访问 ServiceProvider 的 Dispose 方法，而不需要将返回的对象转换为 IDisposable 来调用 Dispose。

**注意：**

如果多次将相同的接口合同添加到服务集合中，最后一个增加的接口合同就会从容器中获得接口。如果需要一些其他功能，比如使用 ASP.NET Core 或 Entity Framework Core 实现的服务，就很容易用不同的实现替换协定。另一方面，对于 ServiceCollection 类，还可以删除服务，并为特定的协定检索所有服务的列表。

接下来，修改 Main()方法来调用 RegisterService 方法，以便在 DI 容器中注册，然后调用 ServiceProvider 的 GetRequiredService 方法，来获取对 HomeController 实例的引用(代码文件 WithDIContainer/Program.cs)：

```
static void Main()
{
  using (ServiceProvider container = RegisterServices())
  {
    var controller = container.GetRequiredService<HomeController>();
    string result = controller.Hello("Katharina");
    Console.WriteLine(result);
  }
}
```

**注意：**

在 ServiceProvider 类中，存在 GetService 和 GetRequiredService 的不同重载。在 ServiceProvider 类中直接实现的方法是带有 Type 参数的 GetService。泛型方法 GetService<t>是一个扩展方法，它采用泛型类型参数，并将其传递给 GetService 方法。

如果该服务在容器中不可用，则 GetService 返回 null。扩展方法 GetRequiredService 检查到 null 结果，如果未找到服务，就抛出一个 InvalidOperationException 异常。如果服务提供程序实现了接口 ISupportsRequiredService，扩展方法 GetRequiredService 将调用提供程序的 GetRequiredService。.NET Core 2.0 的容器没有实现这个接口，但是一些第三方的容器实现了。

启动应用程序时，在 GetRequiredService 方法的请求下，DI 容器将创建 HomeController 类的实例。HomeController 构造函数需要一个实现 IGreetingService 的对象。这个接口也在容器中注册；对于 IGreetingService，需要返回 GreetingService 对象。GreetingService 类有一个默认构造函数，因此容器可以创建一个实例，并将该实例传递给 HomeController 的构造函数。这个实例与控制器变量一起使用，与以前一样用来调用 Hello 方法。

如果不是每个依赖项都在 DI 容器中注册，会发生什么情况？要查看此操作，可以使用 DI 容器删除 IGreetingService 的配置。在这种情况下，容器会抛出 InvalidOperationException 异常。此错误消息显示：在尝试激活 WithDIContainer.HomeController 时无法解析 WithDIContainer.IGreetingService 类型的服务。

## 20.3　服务的生命周期

服务的生命周期定义了服务实例的存在时间。它是否存在于应用程序的生命周期中？每个请求都创建了一个新实例吗？中间还有一些东西，如后面所述。

将服务注册为单例总是返回相同的实例，将服务注册为瞬态，每次注入服务时都会返回一个新对象。还有更多的选择，有更多的问题需要考虑。下面的示例将显示生命周期的特性和问题。该示例还用服务实现了

IDisposable 接口，因此可以看到这是如何处理的。

示例 ServicesLifetime 使用以下 NuGet 包和名称空间：

**包**

Microsoft.Extensions.DependencyInjection

**名称空间**

System

Microsoft.Extensions.DependencyInjection

为了方便地区分不同的实例，每个实例化的服务都指定了一个不同的号码。这个号码是由共享服务创建的。这个共享服务定义了一个简单的接口 INumberService，返回一个号码(代码文件 ServicesLifetime/INumberService.cs)：

```
public interface INumberService
{
  int GetNumber();
}
```

INumberService 的实现总是在 GetNumber 方法中返回一个新号码。该服务注册为一个单例，其号码在其他服务之间共享(代码文件 ServicesLifetime/NumberService.cs)。

```
public class NumberService : INumberService
{
  private int _number = 0;
  public int GetNumber() => Interlocked.Increment(ref _number);
}
```

下面要介绍的其他服务由接口协议 IServiceA、IServiceB、IServiceC 等使用相应的方法 A、B、C 定义。下面的代码片段显示了 IServiceA 的协定(代码文件 ServicesLifetime/IServiceA.cs)：

```
public interface IServiceA
{
  void A();
}
```

在 ServiceA 的实现中，构造函数需要注入 INumberService。通过这个服务，检索号码，将其分配给私有字段_n。在构造函数、方法 A，以及实现 IDisposable 接口的 Dispose 方法中，写入控制台输出，因此可以看到生命周期信息(代码文件 ServicesLifetime/ServiceA.cs)：

```
public class ServiceA : IServiceA, IDisposable
{
  private int _n;
  public ServiceA(INumberService numberService)
  {
    _n = numberService.GetNumber();
    Console.WriteLine($"ctor {nameof(ServiceA)}, {_n}");
  }

  public void A() => Console.WriteLine($"{nameof(A)}, {_n}");

  public void Dispose() =>
    Console.WriteLine($"disposing {nameof(ServiceA)}, {_n}");
}
```

**注意：**
第 17 章详细解释了 IDiposable 接口。

除了服务之外，还实现了控制器 ControllerX。ControllerX 要求构造函数注入三个服务：IServiceA、IServiceB 和 INumberService。在方法 M 中，调用了两个注入的服务。同时，构造函数和 Dispose 信息写入控制台(代码文件 ServicesLifetime/ControllerX.cs)：

```
public class ControllerX : IDisposable
{
  private readonly IServiceA _serviceA;
  private readonly IServiceB _serviceB;
  private readonly int _n;
  private int _countm = 0;
  public ControllerX(IServiceA serviceA, IServiceB serviceB,
```

```
      INumberService numberService)
    {
      _n = numberService.GetNumber();

      Console.WriteLine($"ctor {nameof(ControllerX)}, {_n}");
      _serviceA = serviceA;
      _serviceB = serviceB;
    }

    public void M()
    {
      Console.WriteLine($"invoked {nameof(M)} for the {++_countm}. time");
      _serviceA.A();
      _serviceB.B();
    }

    public void Dispose() => Console.WriteLine(
        $"disposing {nameof(ControllerX)}, {_n}");
  }
```

### 20.3.1 使用单例和临时服务

下面开始注册单例和临时服务。RegisterServices 在方法 SingletonAndTransient 中作为本地函数实现。其中注册了服务 ServiceA、ServiceB、NumberService 和控制器类 ControllerX。NumberService 需要注册为拥有共享状态的单例。ServiceA 也注册为单例。ServiceB 和 ControllerX 是注册的临时服务(代码文件 ServicesLifetime/Program.cs):

```
private static void SingletonAndTransient()
{
  Console.WriteLine(nameof(SingletonAndTransient));

  ServiceProvider RegisterServices()
  {
    IServiceCollection services = new ServiceCollection();
    services.AddSingleton<IServiceA, ServiceA>();
    services.AddTransient<IServiceB, ServiceB>();
    services.AddTransient<ControllerX>();
    services.AddSingleton<INumberService, NumberService>();
    return services.BuildServiceProvider();
  }
  //...
}
```

AddSingleton 和 AddTransient 都是扩展方法，更便于用 Microsoft.Extensions.DependencyInjection 框架注册服务。除了使用这些有用的方法之外，还可以使用 Add 方法注册服务(它本身由方便的方法调用)。Add 方法需要一个包含服务类型、实现类型和服务种类的 ServiceDescriptor。服务的种类使用 ServiceLifetime 枚举类型指定。ServiceLifetime 定义了值 Singleton、Transient 和 Scoped。

```
services.Add(new ServiceDescriptor(typeof(ControllerX),
   typeof(ControllerX), ServiceLifetime.Transient));
```

**注意：**

ServiceCollection 类的 Add 方法是为接口 IServiceCollection 显式实现的。对于这个，只能在使用接口 IServiceCollection 时才能看到方法，使用 ServiceCollection 类型的变量时看不到它。第 4 章中介绍了显式接口的实现。

调用本地函数 RegisterServices 方法来检索 ServiceProvider，获得两次 ControllerX，并调用方法 M，之后释放 ServiceProvider(代码文件 ServicesLifetime/Program.cs):

```
private static void SingletonAndTransient()
{
  //...
  using (ServiceProvider container = RegisterServices())
  {
    ControllerX x = container.GetRequiredService<ControllerX>();
    x.M();
    x.M();
```

```
      Console.WriteLine($"requesting {nameof(ControllerX)}");
      ControllerX x2 = container.GetRequiredService<ControllerX>();
      x2.M();

      Console.WriteLine();
    }
  }
```

**注意：**
第 13 章解释了本地函数。

运行应用程序时，可以看到，请求 ControllerX 时，实例化 ServiceA 和 ServiceB，每次调用 GetNumber 方法时，NumberService 都会返回一个新的数字。当第二次请求 ControllerX 时，不仅新创建了 ControllerX，还创建了 ServiceB，并且注册为暂态。对于 ServiceB，相同的实例会像以前一样使用，不会创建新的实例：

```
SingletonAndTransient
requesting ControllerX
ctor ServiceA, 1
ctor ServiceB, 2
ctor ControllerX, 3
invoked M for the 1. time
A, 1
B, 2
invoked M for the 2. time
A, 1
B, 2
requesting ControllerX
ctor ServiceB, 4
ctor ControllerX, 5
invoked M for the 1. time
A, 1
B, 4

disposing ControllerX, 5
disposing ServiceB, 4
disposing ControllerX, 3
disposing ServiceB, 2
disposing ServiceA, 1
```

### 20.3.2 使用 Scoped 服务

服务也可以在一个作用域内注册。这是介于 transient 和 singleton 之间的服务。对于 singleton，只创建一个实例。每次从容器中请求服务时，transient 都会创建一个新实例。对于 Scoped，总是从相同的作用域中返回相同的实例，但是从不同的作用域中会返回不同的实例。作用域默认用 ASP.NET Core Web 应用程序定义。这里，作用域是一个 HTTP Web 请求。对于 scoped 服务，如果对容器的请求来自同一个 HTTP 请求，则返回相同的实例。而对于不同的 HTTP 请求，会返回其他实例。这允许在 HTTP 请求中轻松共享状态。

对于非 ASP.NET Core Web 应用程序，需要自己创建作用域，以获得 scoped 服务的优势。

下面开始使用本地函数 RegisterServices 注册服务，ServiceA 注册为 scoped 服务，ServiceB 注册为 singleton，ServiceC 注册为 transient(代码文件 ServicesLifetime/Program.cs)：

```
private static void UsingScoped()
{
  Console.WriteLine(nameof(UsingScoped));

  ServiceProvider RegisterServices()
  {
    var services = new ServiceCollection();
    services.AddSingleton<INumberService, NumberService>();
    services.AddScoped<IServiceA, ServiceA>();
    services.AddSingleton<IServiceB, ServiceB>();
    services.AddTransient<IServiceC, ServiceC>();
    return services.BuildServiceProvider();
  }
  //...
}
```

调用 ServiceProvider 的 CreateScope 方法可以创建一个作用域。这将返回实现接口 IServiceScope 的作用域对象，在其中可以访问属于这个作用域的 ServiceProvider，可以在容器中请求服务。在下面的代码片段中，ServiceA 和 ServiceC 被请求两次，而 ServiceB 只请求一次。然后，调用方法 A、B 和 C：

```
private static void UsingScoped()
{
  //...
  using (ServiceProvider container = RegisterServices())
  {
    using (IServiceScope scope1 = container.CreateScope())
    {
      IServiceA a1 = scope1.ServiceProvider.GetService<IServiceA>();
      a1.A();
      IServiceA a2 = scope1.ServiceProvider.GetService<IServiceA>();
      a2.A();
      IServiceB b1 = scope1.ServiceProvider.GetService<IServiceB>();
      b1.B();
      IServiceC c1 = scope1.ServiceProvider.GetService<IServiceC>();
      c1.C();
      IServiceC c2 = scope1.ServiceProvider.GetService<IServiceC>();
      c2.C();
    }

    Console.WriteLine("end of scope1");
  //...
}
```

释放第一个作用域后，就创建另一个作用域。有了第二个作用域，就再次请求同样的服务 ServiceA、ServiceB 和 ServiceC，调用方法：

```
private static void UsingScoped()
{
  //...
  Console.WriteLine("end of scope1");

  using (IServiceScope scope2 = container.CreateScope())
  {
    IServiceA a3 = scope2.ServiceProvider.GetService<IServiceA>();
    a3.A();
    IServiceB b2 = scope2.ServiceProvider.GetService<IServiceB>();
    b2.B();
    IServiceC c3 = scope2.ServiceProvider.GetService<IServiceC>();
    c3.C();
  }
  Console.WriteLine("end of scope2");
  Console.WriteLine();
}
```

运行应用程序时，可以看到，为实例创建了服务，调用了方法，并自动释放它们。当 ServiceA 注册为 scoped 时，在相同的作用域内使用相同的实例。ServiceC 注册为 transient,，因此在这里，为每个对容器的请求创建一个实例。在作用域的末尾，transient 和 scoped 服务会自动释放，但是没有释放 ServiceB，因为 ServiceB 注册为 singleton，需要在作用域的末尾也是存活的：

```
UsingScoped
ctor ServiceA, 1
A, 1
A, 1
ctor ServiceB, 2
B, 2
ctor ServiceC, 3
C, 3
ctor ServiceC, 4
C, 4
disposing ServiceC, 4
disposing ServiceC, 3
disposing ServiceA, 1
end of scope1
```

在第二个作用域的开头，再次实例化 ServiceA 和 ServiceB。请求 ServiceB 时，将返回先前创建的相同对象。在作用域的结尾，再次释放 ServiceA 和 ServiceC。ServiceB 在释放根提供程序后释放：

```
ctor ServiceA, 5
A, 5
```

```
B, 2
ctor ServiceC, 6
C, 6
disposing ServiceC, 6
disposing ServiceA, 5
end of scope2

disposing ServiceB, 2
```

**注意：**

不需要在服务上调用 Dispose 方法来释放它们。使用实现 IDisposable 接口的服务，容器会调用 Dispose 方法。当释放作用域时，将释放 Transient 服务和 scoped 服务。在释放根提供程序时，将释放 Singleton 服务。

在.NET Core 2.0 中，服务实例按照创建的相反顺序来释放。当一个服务需要注入另一个服务时，这一点很重要。例如，ServiceA 要求注入 ServiceB。因此，首先创建 ServiceB，然后创建 ServiceA。在释放时，首先释放 ServiceA，并且在释放过程中仍然可以访问 ServiceB 中的方法。这种行为不同于.NET Core 1.0，.NET Core 1.0 以创建服务实例的顺序释放它们。

### 20.3.3 使用自定义工厂

除了定义使用 transient、scoped 和 singleton 之外，还可以创建自定义工厂或将现有实例传递给容器。下面的代码片段展示了如何实现这一点。

可以使用 AddSingleton 方法的重载版本，将先前创建的实例传递给容器。这里，在 RegisterServices 方法中，首先创建一个 NumberService 对象，然后将其传递给 AddSingleton 方法。使用 GetService 方法，或者在构造函数中注入它，与前面的代码没有什么不同。只需要注意，在本例中，容器不负责调用 Dispose 方法。对于创建并传递到容器的对象，应由开发人员释放这些对象(如果对象需要释放)。还可以使用工厂方法来创建实例，而不是从容器中创建服务。如果服务需要自定义的初始化或定义不受 DI 容器支持的构造函数，这是一个有用的选项。可以通过 IServiceProvider 参数传递委托，并将服务实例返回到 AddSingleton、AddScoped 和 AddTransient 方法。使用示例代码，名为 CreateServiceBFactory 的本地函数返回 ServiceB 对象。如果服务实现的构造函数需要其他服务，则可以使用传递进来的 IServiceProvider 实例检索这些服务(代码文件 ServicesLifetime/Program.cs)：

```
private static void CustomFactories()
{
  Console.WriteLine(nameof(CustomFactories));

  IServiceB CreateServiceBFactory(IServiceProvider provider) =>
    new ServiceB(provider.GetService<INumberService>());

  ServiceProvider RegisterServices()
  {
    var numberService = new NumberService();

    var services = new ServiceCollection();
    services.AddSingleton<INumberService>(numberService);  // add existing

    services.AddTransient<IServiceB>(CreateServiceBFactory);  // use a factory
    services.AddSingleton<IServiceA, ServiceA>();
    return services.BuildServiceProvider();
  }

  using (ServiceProvider container = RegisterServices())
  {
    IServiceA a1 = container.GetService<IServiceA>();
    IServiceA a2 = container.GetService<IServiceA>();
    IServiceB b1 = container.GetService<IServiceB>();
    IServiceB b2 = container.GetService<IServiceB>();
  }
  Console.WriteLine();
}
```

## 20.4　使用选项初始化服务

如前所述，一个服务可以注入另一个服务中。这也可以用来初始化一个带有选项的服务。不能使用服务的构造函数定义非服务协定来进行初始化，因为容器不知道如何初始化它。服务是必要的。但是，为了传递服务的选项，还可以使用已经可用于.NET Core 的服务。

示例 DIWithOptions 使用这些 NuGet 包和名称空间：

**包**

Microsoft.Extensions.DependencyInjection

Microsoft.Extensions.Options

**名称空间**

System

Microsoft.Extensions.DependencyInjection

Microsoft.Extensions.Options

示例代码使用之前使用的 GreetingService 进行修改，以传递选项。服务所需的配置值由类 GreetingServiceOptions 定义。样例代码需要一个带有 From 属性的 String 参数(代码文件 DIWithOptions/GreetingServiceOptions.cs)：

```
public class GreetingServiceOptions
{
  public string From { get; set; }
}
```

可以指定带有 IOptions<T>参数的构造函数，来传递服务的选项。前面定义的类 GreetingServiceOptions 是用于 IOptions 的泛型类型。传递给构造函数的值用于初始化字段_from(代码文件 DIWithOptions/GreetingService.cs)：

```
public class GreetingService : IGreetingService
{
  public GreetingService(IOptions<GreetingServiceOptions> options) =>
    _from = options.Value.From;

  private readonly string _from;

  public string Greet(string name) => $"Hello, {name}! Greetings from {_from}";
}
```

> **注意：**
> IOptions 接口和用于选项的服务在 NuGet 包 Microsoft.Extensions.Options 中实现。

为了便于使用 DI 容器注册服务，定义了扩展方法 AddGreetingService。该方法扩展了 IServiceCollection 接口，并允许通过委托传递 GreetingServiceOptions。在实现代码中，Configure 方法用于通过 IOptions 接口指定配置。Configure 方法是 NuGet 包 Microsoft.Extensions.Options 中 IServiceCollection 的扩展方法(代码文件 DIWithOptions/GreetingServiceExtensions.cs)：

```
public static class GreetingServiceExtensions
{
  public static IServiceCollection AddGreetingService(
    this IServiceCollection collection,
    Action<GreetingServiceOptions> setupAction)
  {
    if (collection == null)
      throw new ArgumentNullException(nameof(collection));
    if (setupAction == null)
      throw new ArgumentNullException(nameof(setupAction));

    collection.Configure(setupAction);
    return collection.AddTransient<IGreetingService, GreetingService>();
  }
}
```

通过构造函数注入使用 GreetingService 的 HomeController 不需要任何更改(代码文件 DIWithOptions/HomeController.cs)：

```
public class HomeController
{
  private readonly IGreetingService _greetingService;
  public HomeController(IGreetingService greetingService)
  {
    _greetingService = greetingService;
  }

  public string Hello(string name) => _greetingService.Greet(name);
}
```

现在可以使用辅助方法 AddGreetingService 注册服务。GreetingService 的配置是通过传递所需选项来完成的。还需要一个实现 IOptions 接口的服务。在这里，可以使用扩展方法 AddOptions。该方法添加了几个接口，并将其映射到与选项一起使用的实现(代码文件 DIWithOptions/Program.cs)：

```
static ServiceProvider RegisterServices()
{
  var services = new ServiceCollection();
  services.AddOptions();
  services.AddGreetingService(options =>
  {
    options.From = "Christian";
  });
  services.AddTransient<HomeController>();
  return services.BuildServiceProvider();
}
```

该服务现在可以像以前一样使用。HomeController 从容器中检索，在使用 IGreetingService 的 HomeController 中使用构造函数注入：

```
static void Main()
{
  using (var container = RegisterServices())
  {
    var controller = container.GetService<HomeController>();
    string result = controller.Hello("Katharina");
    Console.WriteLine(result);
  }
}
```

运行应用程序时，现在可以使用以下选项：

```
Hello, Katharina! Greetings from Christian
```

## 20.5 使用配置文件

需要在配置文件中配置服务时，也可以使用前面所示的选项。然而，有一种更直接的方法：可以使用.NET 配置特性和对选项的扩展。使用 NuGet 包 Microsoft.Extensions.Options.ConfigurationExtensions 中的配置可以扩展选项。

样例 DIWithConfiguration 使用如下 NuGet 包和名称空间：

**包**

Microsoft.Extensions.Configuration

Microsoft.Extensions.Configuration.Json

Microsoft.ExtensionsDependencyInjection

Microsoft.Extensions.Options

Microsoft.Extensions.Options.ConfigurationExtensions

**名称空间**

System

Microsoft.Extensions.Configuration

Microsoft.Extensions.DependencyInjection

Microsoft.Extensions.Options

示例代码基于前一节的示例，但是现在可以使用配置扩展选项。不需要更改 GreetingService 类，它仍然使用 IOptions 接口进行初始化。更改的是 AddGreetingService 扩展方法，这就更容易使用该服务。该方法的第二个参数现在是 IConfiguration 类型，以接收配置值。config 参数用于将其传递给 Configure 扩展方法。Configure 扩展方法与前面使用的方法不同；这个方法在 NuGet 包 Microsoft.Extensions.Options.ConfigurationExtensions 中定义(代码文件 DIWithConfiguration/GreetingServiceExtensions.cs)：

```
public static class GreetingServiceExtensions
{
  public static IServiceCollection AddGreetingService(
    this IServiceCollection collection, IConfiguration config)
  {
    if (collection == null)
      throw new ArgumentNullException(nameof(collection));
    if (config == null) throw new ArgumentNullException(nameof(config));

    collection.Configure<GreetingServiceOptions>(config);
    return collection.AddTransient<IGreetingService, GreetingService>();
  }
}
```

可以从环境变量、程序参数和不同格式(如 XML、INI 和 JSON 文件)的文件中读取配置。这里，用于读取 JSON 配置的提供程序可以添加 NuGet 包 Microsoft.Extensions.Configuration 和 Microsoft.Extensions.Json。在方法 DefineConfiguration 中，首先创建一个 ConfigurationBuilder，然后使用 fluent API 为可以读取 JSON 文件的目录配置基本路径，并配置 JSON 文件本身。文件 appsettings.json 用于读取配置。在 ConfigurationBuilder 设置之后，调用 Build 方法来返回一个对象，该对象实现了可访问配置值的 IConfiguration(代码文件 DIWithConfiguration/Program.cs)：

```
static void DefineConfiguration()
{
  IConfigurationBuilder configBuilder = new ConfigurationBuilder()
    .SetBasePath(Directory.GetCurrentDirectory())
    .AddJsonFile("appsettings.json");
    Configuration = configBuilder.Build();
}

public static IConfiguration Configuration { get; set; }
```

配置文件指定 GreetingService 配置的 From 设置(配置文件 DIWithConfiguration/appsettings.json)：

```
{
  "GreetingService": {
    "From": "Matthias"
  }
}
```

**注意：**

.NET 配置详见第 30 章。

ServiceCollection 的配置与以前一样。还需要指定 IOptions 接口。不同的是，AddGreetingService 扩展方法的新版本传递了 IConfiguration 值。这可以通过访问 Configuration 属性来完成，该属性以前定义为读取 GreetingService 部分，它传递包含该部分的值(代码文件 DIWithConfiguration/Program.cs)：

```
static ServiceProvider RegisterServices()
{
  var services = new ServiceCollection();
  services.AddOptions();
  services.AddSingleton<IGreetingService, GreetingService>();
  services.AddGreetingService(
    Configuration.GetSection("GreetingService"));
  services.AddTransient<HomeController>();
  return services.BuildServiceProvider();
}
```

在注册服务之前，需要指定配置。这在 Main()方法中完成，如下所示。运行应用程序时，它的行为就像以前一样，但是配置来自一个文件：

```
static void Main()
{
  DefineConfiguration();
  var container = RegisterServices();
  var controller = container.GetService<HomeController>();
  string result = controller.Hello("Katharina");
  Console.WriteLine(result);
}
```

# 20.6 创建平台独立性

依赖注入也可在平台独立的库中使用特定于平台的特性。例如，可以创建一个用于 WPF、UWP 和 Xamarin 应用程序的.NET 标准库，并将调用转换为特定于平台的 API。调用 Web API 可以实现，因此它是一个完全独立的平台。如打开消息对话框这样简单的事情，是特定于平台的。

**注意：**
MVVM 模式详见第 34 章。UWP 应用程序的编写详见第 33~36 章，用 Xamarin 编写应用程序详见第 37 章。

下一个示例解决方案 PlatformIndependenceSample 由表 20-1 中的项目组成。

表 20-1

| | |
|---|---|
| DISampleLib | 这是一个由服务、服务协定和视图模型组成的.NET 标准库。由于消息服务是特定于平台的，因此对于此服务，只在库中定义了一个协定(IMessageService) |
| WPFClient | WPF 客户端应用程序引用了 DISampleLib，并包含了对 NuGet 包 Microsoft.Extensions.DependencyInjection 的引用。这个应用程序使用 XAML 定义了一个用户界面，并利用了.NET 标准库中的视图模型和服务。对于接口 IMessageService，服务 WPFMessageService 在 WPF 项目中实现 |
| UWPClient | UWP 客户应用程序类似于 WPF 应用程序。它只是需要一个不同的 IMessageService 实现：UWPMessageService |
| XamarinClient | 使用 Xamarin 创建 Xamarin.Forms 应用程序。这会得到一个用于 Android 的项目、一个用于 iOS 的项目和一个用于 UWP 的项目。公共代码(可以共享用户界面)在一个共享项目中。对于每个 Xamarin.Forms 项目，需要引用.NET 标准库和 NuGet 包 Microsoft.Extensions.DependencyInjection。在共享项目中，用户界面的代码、DI 容器的设置以及 IMessageService 接口的实现 XamarinMessageService 在 UWP、Android 和 iPhone 是通用的 |

**注意：**
为了在 Visual Studio 中使用 Xamarin 项目模板，需要使用 Visual Studio 安装程序安装通过.NET 开发的工作负载 Mobile。也可以使用 Visual Studio for Mac。为了成功地编译 iOS，还需要 Mac。

## 20.6.1 .NET 标准库

下面从实现服务和协定的.NET 标准库开始。接口 IMessageAsync 是一个协定。该协定定义了 ShowMessageAsync 方法，调用该方法时应该显示弹出式窗口。如前所述，在.NET 标准中创建这些对话框是不可能的(代码文件 PlatformIndependenceSample/DISampleLib/IMessageService.cs)：

```
public interface IMessageService
{
  Task ShowMessageAsync(string message);
}
```

接口 IMessageService 在类 ShowMessageViewModel 中使用。实现该接口的对象将通过

ShowMessageViewModel 的构造函数注入。这个视图模型类定义了从特定平台的用户界面中触发的命令 ShowMessageCommand。在执行此命令时，将调用 IMessageService 的 ShowMessageAsync 方法(代码文件 PlatformIndependenceSample/DISampleLib/ShowMessageViewModel.cs)：

```
public class ShowMessageViewModel
{
  private readonly IMessageService _messageService;
  public ShowMessageViewModel(IMessageService messageService)
  {
    _messageService = messageService ??
      throw new ArgumentNullException(nameof(messageService));

    ShowMessageCommand = new RelayCommand(ShowMessage);
  }

  public ICommand ShowMessageCommand { get; }

  public void ShowMessage()
  {
    _messageService.ShowMessageAsync("A message from the view-model");
  }
}
```

## 20.6.2　WPF 应用程序

使用该库的第一个客户应用程序是 WPF 应用程序。WPFMessageService 类实现了 IMessageService 接口。在 WPF 中，可以使用 MessageBox.Show 打开对话框。此类不提供异步功能；因此，在 ShowMessageAsync 方法中返回完成的任务，以履行协定 (代码文件 PlatformIndependenceSample/WPFClient/WPFMessageService.cs)：

```
public class WPFMessageService : IMessageService
{
  public Task ShowMessageAsync(string message)
  {
    MessageBox.Show(message);
    return Task.CompletedTask;
  }
}
```

在 App 类中，创建了 DI 容器，并注册了服务。WPFMessageService 类映射到 IMessageService 接口上(代码文件 PlatformIndependenceSample/WPFClient/App xaml.cs)：

```
protected override void OnStartup(StartupEventArgs e)
{
  base.OnStartup(e);
  RegisterServices();
}

public void RegisterServices()
{
  var services = new ServiceCollection();
  services.AddSingleton<IMessageService, WPFMessageService>();
  services.AddTransient<ShowMessageViewModel>();
  Container = services.BuildServiceProvider();
}

public IServiceProvider Container { get; private set; }
```

在用户界面的代码隐藏文件中，ViewModel 属性是通过请求容器中的 ShowMessageViewModel 来设置的(代码文件 PlatformIndependenceSample/WPFClient/MainWindow.xaml.cs)：

```
public MainWindow()
{
  InitializeComponent();
  ViewModel = (Application.Current as App)
    .Container.GetService<ShowMessageViewModel>();
  this.DataContext = this;
}

public ShowMessageViewModel ViewModel { get; }
```

在 XAML 代码中，Button 元素定义了一个 Command 属性，该属性绑定到视图模型的 ShowMessageCommand

上 (XAML 文件 PlatformIndependenceSample/WPFClient/MainWindow.xaml)：

```
<Button Content="Click Me!"
  Command="{Binding ViewModel.ShowMessageCommand, Mode=OneTime}" />
```

运行应用程序时，单击按钮将调用从容器中请求的视图模型中的命令，而命令处理程序将调用一个服务，该服务由 WPF 应用程序实现，以显示如图 20-1 所示的 MessageBox。

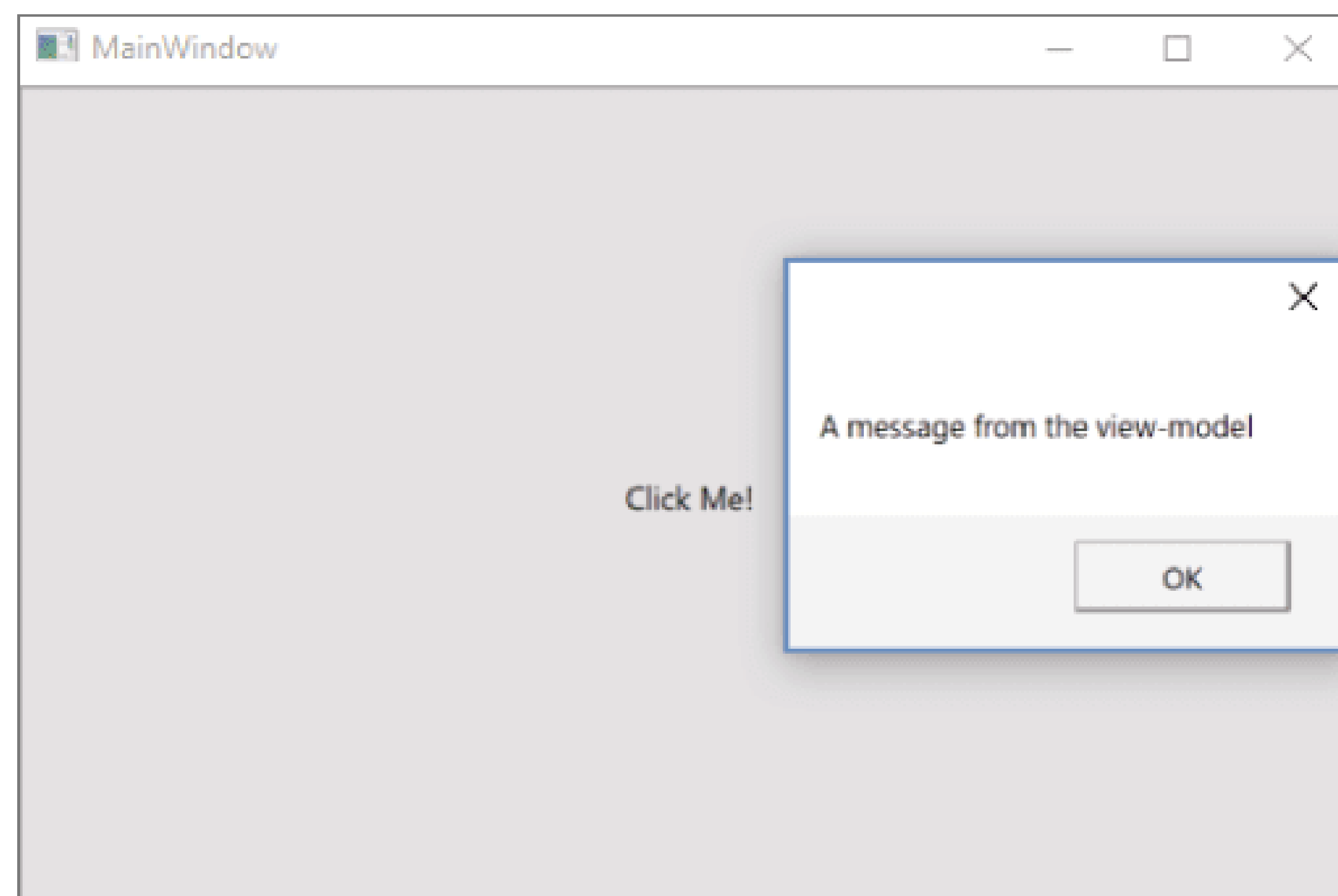



图 20-1

### 20.6.3 UWP 应用程序

UWP 应用程序与 WPF 应用程序非常相似，但是有一些重要的区别。第一个区别是通过 UWPMessageService 类呈现的；要显示对话框，使用 MessageDialog 类。这个类提供了异步方法 ShowAsync(代码文件 PlatformIndependenceSample/UWPClient/UWPMessageService.cs)：

```
public class UWPMessageService : IMessageService
{
  public async Task ShowMessageAsync(string message) =>
    await new MessageDialog(message).ShowAsync();
}
```

DI 容器的填充方式相同，但 UWPMessageSerivce 现在用来履行 IMessageService 接口的协定(代码文件 PlatformIndependenceSample/UWPClient/App.xaml.cs)：

```
public void RegisterServices()
{
  var services = new ServiceCollection();
  services.AddSingleton<IMessageService, UWPMessageService>();
  services.AddTransient<ShowMessageViewModel>();
  Container = services.BuildServiceProvider();
}

public IServiceProvider Container { get; private set; }
```

请求容器中视图模型的方式与 WPF 相同(代码文件 PlatformIndependenceSample/UWPClient /MainPage.xaml.cs)：

```
public MainPage()
{
  this.InitializeComponent();
  ViewModel = (Application.Current as App)
    .Container.GetService<ShowMessageViewModel>();
}

public ShowMessageViewModel ViewModel { get; }
```

与 UWP 的另一个区别是使用编译绑定，但这与依赖注入并不真正相关(XAML 文件 PlatformIndependenceSample/UWPClient/MainPage.xaml)：

```
<Button Content="Click Me!"
  Command="{x:Bind ViewModel.ShowMessageCommand, Mode=OneTime}" />
```

运行应用程序时，可以看到相同的行为，但使用的是 UWP 用户界面(见图 20-2)。

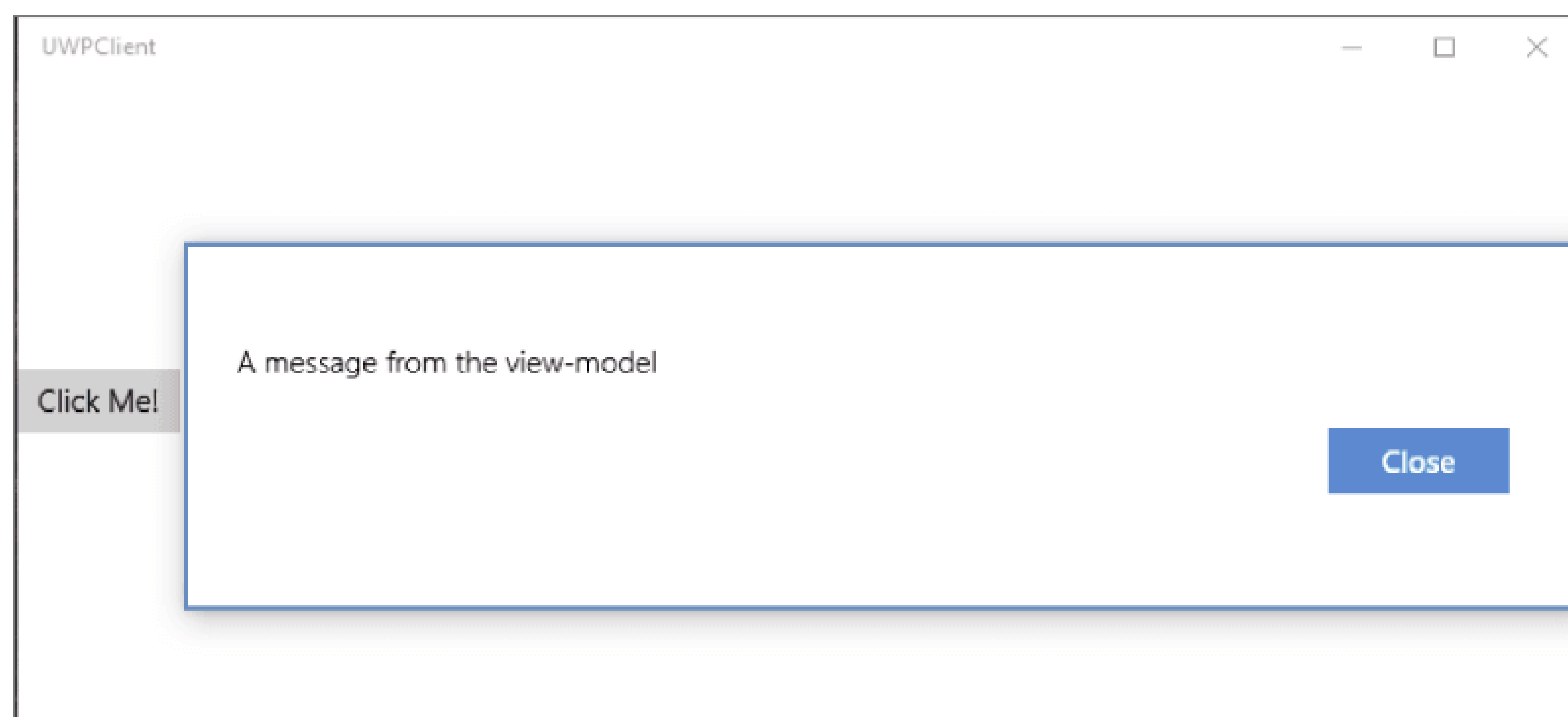



图 20-2

### 20.6.4　Xamarin 应用程序

使用 Xamarin.Forms 应用程序实现相同的特性时，依赖注入还有一个有趣的方法。在 Xamarin.Forms 中，显示消息对话框的方法需要 Page 对象，因此实现 IMessageService 的 XamarinMessageService 需要配置为接收 Page 对象。不可能更改 IMessageService，因为 Page 类型无法用于.NET 标准库。通过 XamarinMessageService 类来使用额外的配置属性也是不可能的，因为这个类是在视图模型类型中实例化，而视图模型也是在.NET 标准库中实现的。一个好方法是使用只在 Xamarin.Forms 应用程序中使用的服务协定添加另一个服务。XamarinMessageService 构造函数需要一个实现 IPageService 接口的对象。然后使用此服务检索 Page，通过它就可以调用 DisplayAlert 方法(代码文件 PlatformIndependenceSample/XamarinClient/XamarinMessageService.cs)：

```
public class XamarinMessageService : IMessageService
{
  private readonly IPageService _pageService;
  public XamarinMessageService(IPageService pageService)
  {
    _pageService = pageService;
  }

  public Task ShowMessageAsync(string message)
  {
    return _pageService.Page.DisplayAlert("Message", message, "Close");
  }
}
```

协定 IPageService 只是根据需要为对话框定义了 Page 属性 (代码文件 PlatformIndependenceSample/XamarinClient/IPageService.cs)：

```
public interface IPageService
{
  Page Page { get; set; }
}
```

IPageService 的实现只是一个简单自动属性(代码文件 PlatformIndependenceSample/XamarinClient/PageService.cs)：

```
public class PageService : IPageService
{
  public Page Page { get; set; }
}
```

容器是在 App 类中创建的。这次，XamarinMessageSerivce 映射为 IMessageService 协定的实现，也需要列出 IPageService (代码文件 PlatformIndependenceSample/XamarinClient/App.xaml.cs)：

```
public App()
{
  InitializeComponent();
  RegisterServices();
```

```
  MainPage = new XamarinClient.MainPage();
}

public void RegisterServices()
{
  var services = new ServiceCollection();
  services.AddSingleton<IPageService, PageService>();
  services.AddSingleton<IMessageService, XamarinMessageService>();
  services.AddTransient<ShowMessageViewModel>();
  Container = services.BuildServiceProvider();
}

public IServiceProvider Container { get; private set; }
```

在代码隐藏文件中，Page 属性需要与 PageService 相关联(代码文件 PlatformIndependenceSample/XamarinClient/MainPage.xaml.cs)：

```
public MainPage()
{
  InitializeComponent();
  IServiceProvider container = (Application.Current as App).Container;
  ViewModel = container.GetService<ShowMessageViewModel>();
  this.BindingContext = this;
  container.GetService<IPageService>().Page = this;
}

public ShowMessageViewModel ViewModel { get; }
```

XAML 代码与前面的示例一样(XAML 文件 PlatformIndependenceSample/XamarinClient/MainPage.xaml)：

```
<Button Text="Click Me!"
  Command="{Binding ViewModel.ShowMessageCommand, Mode=OneWay}" />
```

运行应用程序时，会弹出警告消息，如图 20-3 所示。

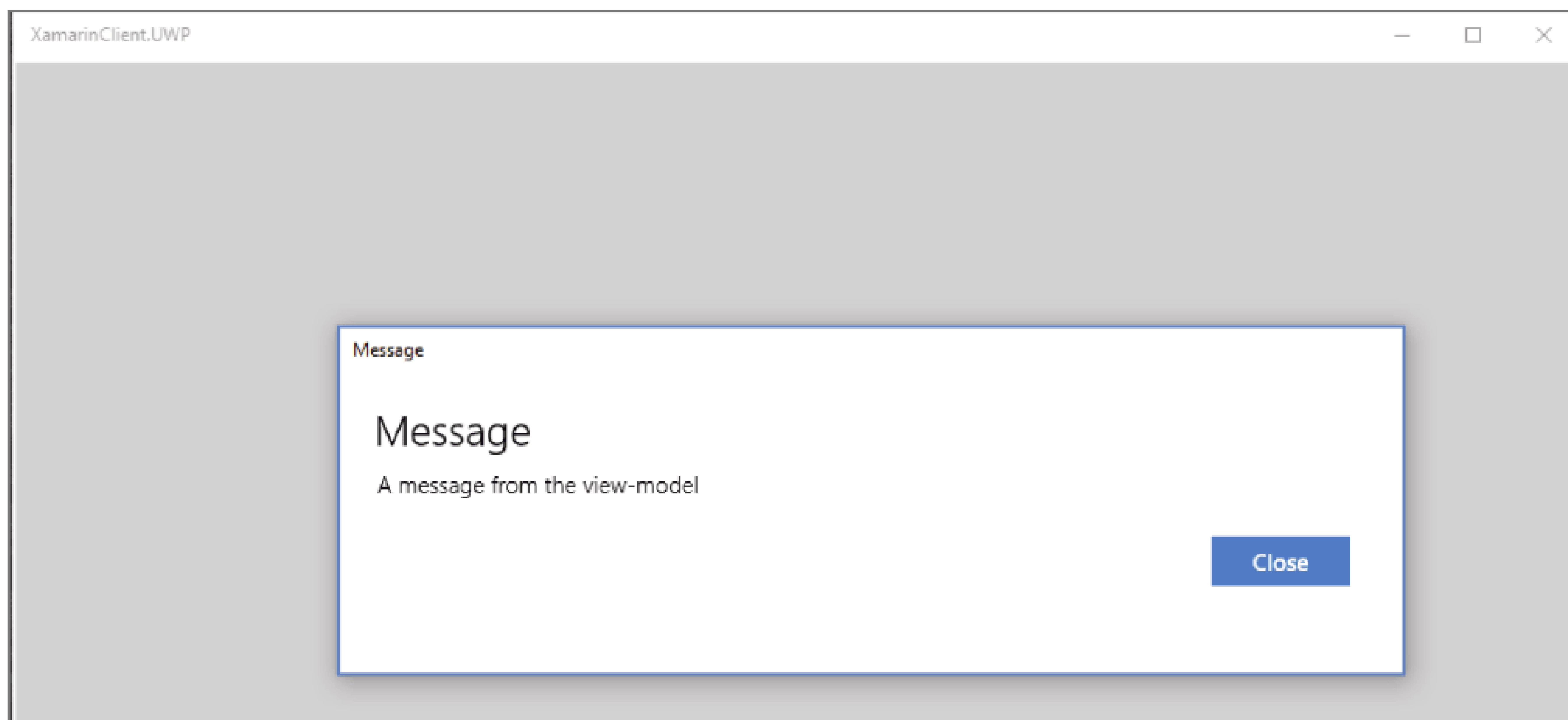



图 20-3

## 20.7 使用其他 DI 容器

Microsoft.Extensions.DependencyInjection 是一个简单的 DI 容器；许多第三方的容器都提供了额外的功能。例如，Autofac 允许在配置文件中配置服务。

ASP.NET Core 使用 Microsoft.Extensions.DependencyInjection，除此之外，还可以使用适配器配置它——使用其他第三方依赖注入容器，如 Autofac、Rezolver、Scan、Neleus、CuteAnt、fm、Dryloc 、CuteAnt、Stashbox 等。只需要以 NuGet 包的形式添加一个适配器，根据容器的需求进行初始化。

示例项目 DIWithAutofac 使用和之前的实现相同的 HomeController 和 GreetingService，但它使用 Autofac 依

赖注入容器适配器。为此，需要如下 NuGet 包和名称空间：

**包**

Autofac.Extensions.DependencyInjection

Microsoft.ExtensionsDependencyInjection

**名称空间**

Autofac

Autofac.Extensions.DependencyInjection

Microsoft.Extensions.DependencyInjection

System

为了使用 Autofac 容器适配器，服务应与以前一样，在 ServiceCollection 中注册。现在不用创建 IServiceProvider，而是使用来自 Autofac 的 ContainerBuilder。可以使用调用 Populate 方法的 ServiceCollection 来填充该构建器。这种方法可以给这个容器添加数百个 ASP.NET Core 服务。这个容器还支持一些 Register 方法来添加托管服务。Build 方法现在创建一个容器，并使用 IContainer 接口返回它(代码文件 DIWithAutoFac/Program.cs)：

```
static IContainer RegisterServices()
{
  var services = new ServiceCollection();
  services.AddSingleton<IGreetingService, GreetingService>();
  services.AddTransient<HomeController>();

  var builder = new ContainerBuilder();
  builder.Populate(services);
  return builder.Build();
}
```

现在可以使用 Resolve 方法来解析这个容器中的服务。除此之外，不需要任何改变。HomeController 通过依赖注入接收 GreetingService(代码文件 DIWithAutoFac/Program.cs)：

```
static void Main()
{
  using (IContainer container = RegisterServices())
  {
    var controller = container.Resolve<HomeController>();
    string result = controller.Hello("Katharina");
    Console.WriteLine(result);
  }
}
```

## 20.8　小结

本章介绍了依赖注入，继续讨论了使用微软容器 Microsoft.Extensions.DependencyInjection 的各种场景，包括通过选项的配置和.NET Core 配置。还讨论了如何通过特定于平台的特性与 WPF、UWP、Xamarin，为不同的平台使用依赖注入。

在本书的几个章节中，依赖注入具有重要的作用。第 26 章展示了如何使用依赖注入与 Entity Framework Core 以及如何取代内置功能。第 28 章探讨了如何将依赖注入用于单元测试和不应该通过单元测试进行测试的模拟功能。第 30 章和后续的章节介绍如何使用依赖注入与 ASP.NET Core，如何在容器中注册数以百计的服务。第 34 章显示了 MVVM 模式和用于 XAML 应用程序的其他模式。在这些应用程序中，依赖注入是一个重要的基础。

第 21 章详细讨论 Task 和 Parallel 类的并行编程，以帮助在操作系统中使用多个核心。还要阐述使用多个任务时出现的问题。

第 21 章

# 任务和并行编程

**本章要点**

- 多线程概述
- 使用 Parallel 类
- 使用任务
- 使用取消架构
- 使用数据流库
- 使用计时器
- 理解线程问题
- 使用 lock 关键字
- 用监视器同步
- 用互斥同步
- 使用 Semaphore 和 SemaphoreSlim
- 使用 ManualResetEvent、AutoResetEvent 和 CountdownEvent
- 处理障碍
- 用 ReaderWriterLockSlim 管理读取器和写入器

**本章源代码下载地址(wrox.com)：**

打开 www.wrox.com 的 Download Code 选项卡可下载本章源代码。源代码也可以在 Tasks 目录的 https://github.com/ProfessionalCSharp/ProfessionalCSharp7 中找到。

- Parallel
- Task
- Cancellation
- DataFlow
- Timer
- WinAppTimer
- ThreadingIssues
- SynchronizationSamples

- BarrierSample
- ReaderWriterLockSample

## 21.1　概述

使用多线程有几个原因。假设从应用程序进行网络调用需要一定的时间。我们不希望用户界面停止响应，让用户一直等待，直到从服务器返回一个响应。用户可以同时执行其他一些操作，或者甚至取消发送给服务器的请求。这些都可以使用线程来实现。

对于所有需要等待的操作，例如，因为文件、数据库或网络访问都需要一定的时间，此时就可以启动一个新线程，同时完成其他任务。即使是处理密集型的任务，线程也是有帮助的。一个进程的多个线程可以同时运行在不同的 CPU 上，或多核 CPU 的不同内核上。

还必须注意运行多线程时的一些问题。它们可以同时运行，但如果线程访问相同的数据，就很容易出问题。为了避免出问题，必须实现同步机制。

.NET 提供了线程的一个抽象机制：任务。任务允许建立任务之间的关系，例如，第一个任务完成时，应该继续下一个任务。也可以建立一个层次结构，其中包含多个任务。

除了使用任务之外，还可以使用 Parallel 类实现并行活动。需要区分数据并行(在不同的任务之间同时处理一些数据)和任务并行性(同时执行不同的功能)。

在创建并行程序时，有很多不同的选择。应该使用适合场景的最简单选项。本章首先介绍 Parallel 类，它提供了非常简单的并行性。如果这就是需要的类，使用这个类即可。如果需要更多的控制，比如需要管理任务之间的关系，或定义返回任务的方法，就要使用 Task 类。

本章还包括数据流库，如果需要基于操作的编程通过管道传送数据，这可能是最简单的一个库了。

如果需要更多地控制并行性，如设置优先级，就需要使用 Thread 类。

**注意：**

通过关键字 async 和 await 来使用异步方法参见第 15 章。Parallel LINQ 提供了任务并行性的一种变体，详见第 12 章。

创建一个并发执行多个任务的程序，可能导致争用条件和死锁。需要注意同步技术。

要避免同步问题，最好不要在线程之间共享数据。当然，这并不总是可行的。如果需要共享数据，就必须使用同步技术，确保一次只有一个线程访问和改变共享状态。如果不注意同步，就会出现争用条件和死锁。一个主要问题是错误会不时地发生。如果 CPU 核心比较多，错误数量就会增加。这些错误通常很难找到。所以最好从一开始就注意同步。

使用多个任务是很容易的，只要它们不访问相同的变量。在某种程度上可以避免这种情况，但有时，一些数据需要共享。共享数据时，需要应用同步技术。线程访问相同的数据，而没有进行同步，立即出现问题是比较幸运的。但很少会出现这种情况。本章讨论了争用条件和死锁，以及如何应用同步机制来避免它们。

.NET Framework 提供了同步的几个选项。同步对象可以用在一个进程中或跨进程中。可以使用它们来同步一个任务或多个任务来访问一个资源或许多资源。同步对象也可以用来通知完成的任务。本章介绍所有这些同步对象。

**注意：**

尽可能使用不可变的类型可能部分地避免同步问题。在不可变的数据结构中，数据只能初始化，以后就不能更改了。所以这些类型不需要同步。

在长长的概述之后，下面从 Parallel 类开始——给应用程序添加并行性的一种简单方式。

## 21.2 Parallel 类

Parallel 类是对线程的一个很好的抽象。该类位于 System.Threading.Tasks 名称空间中，提供了数据和任务并行性。

Parallel 类定义了并行的 for 和 foreach 的静态方法。对于 C#的 for 和 foreach 语句而言，循环从一个线程中运行。Parallel 类使用多个任务，因此使用多个线程来完成这个作业。

Parallel.For()和 Parallel.ForEach()方法在每次迭代中调用相同的代码，而 Parallel.Invoke()方法允许同时调用不同的方法。Parallel.Invoke()用于任务并行性，而 Parallel.ForEach()用于数据并行性。

### 21.2.1 使用 Parallel.For()方法循环

Parallel.For()方法类似于 C#的 for 循环语句，也是多次执行一个任务。使用 Parallel.For()方法，可以并行运行迭代。迭代的顺序没有定义。

ParallelSamples 的示例代码使用了如下名称空间：

**名称空间**

System

System.Linq

System.Threading

System.Threading.Tasks

> **注意：**
> 这个示例使用命令行参数。为了了解不同的特性，应在启动示例应用程序时传递不同的参数，如下所示，或检查 Main()方法。在 Visual Studio 中，可以在项目属性的 Debug 选项中传递命令行参数。使用 dotnet 命令行，传递命令行参数-p，则可以启动命令 dotnet run -- - p。

有关线程和任务的信息，下面的 Log 方法把线程和任务标识符写到控制台(代码文件 ParallelSamples/Program.cs)：

```
public static void Log(string prefix) =>
  Console.WriteLine($"{prefix}, task: {Task.CurrentId}, " +
    $"thread: {Thread.CurrentThread.ManagedThreadId}");
```

下面看看在 Parallel.For()方法中，前两个参数定义了循环的开头和结束。示例从 0 迭代到 9。第 3 个参数是一个 Action<int>委托。整数参数是循环的迭代次数，该参数被传递给委托引用的方法。Parallel.For()方法的返回类型是 ParallelLoopResult 结构，它提供了循环是否结束的信息。

```
public static void ParallelFor()
{
  ParallelLoopResult result =
    Parallel.For(0, 10, i =>
    {
      Log($"S {i}");
      Task.Delay(10).Wait();
      Log($"E {i}");
    });
  Console.WriteLine($"Is completed: {result.IsCompleted}");
}
```

在 Parallel.For()的方法体中，把索引、任务标识符和线程标识符写入控制台中。从输出可以看出，顺序是不能保证的。如果再次运行这个程序，可以看到不同的结果。程序这次的运行顺序是 2-4-0-6-8，有 9 个任务和 6 个线程。任务不一定映射到一个线程上。线程也可以被不同的任务重用。

```
S 2 task: 1, thread: 3
S 4 task: 2, thread: 4
S 0 task: 4, thread: 2
S 6 task: 5, thread: 5
S 8 task: 3, thread: 6
```

```
E 6 task: 5, thread: 5
E 0 task: 4, thread: 2
S 1 task: 4, thread: 2
E 4 task: 2, thread: 4
S 5 task: 8, thread: 4
E 2 task: 1, thread: 3
S 9 task: 9, thread: 3
E 8 task: 3, thread: 6
S 3 task: 7, thread: 8
S 7 task: 6, thread: 5
E 5 task: 8, thread: 4
E 1 task: 4, thread: 2
E 3 task: 7, thread: 8
E 7 task: 6, thread: 5
E 9 task: 9, thread: 3
Is completed: True
```

并行体内的延迟等待 10 毫秒，会有更好的机会来创建新线程。如果删除这行代码，就会使用更少的线程和任务。

在结果中还可以看到，循环的每个 end-log 使用与 start-log 相同的线程和任务。使用 Task.Delay()和 Wait()方法会阻塞当前线程，直到延迟结束。

修改前面的示例，现在使用 await 关键字和 Task.Delay()方法(代码文件 ParallelSamples/Program.cs)：

```
public static void ParallelForWithAsync()
{
  ParallelLoopResult result =
    Parallel.For(0, 10, async i =>
    {
      Log($"S {i}");
      await Task.Delay(10);
      Log($"E {i}");
    });
  Console.WriteLine($"is completed: {result.IsCompleted}");
}
```

其结果如以下代码片段所示。在输出中可以看到，调用 Thread.Delay()方法后，线程发生了变化。例如，循环迭代 8 在延迟前的线程 ID 为 7，在延迟后的线程 ID 为 5。在输出中还可以看到，任务不再存在，只有线程了，而且这里重用了前面的线程。另一个重要的方面是，Parallel 类的 For()方法并没有等待延迟，而是直接完成。Parallel 类只等待它创建的任务，而不等待其他后台活动。在延迟后，也有可能完全看不到方法的输出，出现这种情况的原因是主线程(是一个前台线程)结束，所有的后台线程被终止。下一章将讨论前台线程和后台线程。

```
S 0, task: 5, thread: 1
S 8, task: 8, thread: 7
S 6, task: 7, thread: 8
S 4, task: 9, thread: 6
S 2, task: 6, thread: 5
S 7, task: 7, thread: 8
S 1, task: 5, thread: 1
S 5, task: 9, thread: 6
S 9, task: 8, thread: 7
S 3, task: 6, thread: 5
Is completed: True
E 2, task: , thread: 8
E 0, task: , thread: 8
E 8, task: , thread: 5
E 6, task: , thread: 7
E 4, task: , thread: 6
E 5, task: , thread: 7
E 7, task: , thread: 7
E 1, task: , thread: 6
E 3, task: , thread: 5
E 9, task: , thread: 8
```

**注意：**

从这里可以看到，虽然使用.NET 和 C#的异步功能十分方便，但是知道后台发生了什么仍然很重要，而且必须留意一些问题。

### 21.2.2 提前中断 Parallel.For

也可以提前中断 Parallel.For()方法，而不是完成所有迭代。For()方法的一个重载版本接受 Action<int, ParallelLoopState>类型的第 3 个参数。使用这些参数定义一个方法，就可以调用 ParallelLoopState 的 Break()或 Stop()方法，以影响循环的结果。

注意，迭代的顺序没有定义(代码文件 ParallelSamples/Program.cs)。

```
public static void StopParallelForEarly()
{
  ParallelLoopResult result =
    Parallel.For(10, 40, (int i, ParallelLoopState pls) =>
    {
      Log($"S {i}");
      if (i > 12)
      {
        pls.Break();
        Log($"break now... {i}");
      }
      Task.Delay(10).Wait();
      Log($"E {i}");
    });
  Console.WriteLine($"Is completed: {result.IsCompleted}");
  Console.WriteLine($"lowest break iteration: {result.LowestBreakIteration}");
}
```

应用程序的这次运行说明，迭代在值大于 12 时中断，但其他任务可以同时运行，有其他值的任务也可以运行。在中断前开始的所有任务都可以继续运行，直到结束。利用 LowestBreakIteration 属性，可以忽略其他不需要的任务的结果。

```
S 31, task: 6, thread: 8
S 17, task: 7, thread: 5
S 10, task: 5, thread: 1
S 24, task: 8, thread: 6
break now 24, task: 8, thread: 6
S 38, task: 9, thread: 7
break now 38, task: 9, thread: 7
break now 31, task: 6, thread: 8
break now 17, task: 7, thread: 5
E 17, task: 7, thread: 5
E 10, task: 5, thread: 1
S 11, task: 5, thread: 1
E 38, task: 9, thread: 7
E 24, task: 8, thread: 6
E 31, task: 6, thread: 8
E 11, task: 5, thread: 1
S 12, task: 5, thread: 1
E 12, task: 5, thread: 1
S 13, task: 5, thread: 1
break now 13, task: 5, thread: 1
E 13, task: 5, thread: 1
Is completed: False
lowest break iteration: 13
```

### 21.2.3 Parallel.For()方法的初始化

Parallel.For()方法使用几个线程来执行循环。如果需要对每个线程进行初始化，就可以使用 Parallel.For<TLocal>()方法。除了 from 和 to 对应的值之外，For()方法的泛型版本还接受 3 个委托参数。第一个参数的类型是 Func<TLocal>。因为这里的例子对于 TLocal 使用字符串，所以该方法需要定义为 Func<string>，即返回 string 的方法。这个方法仅对用于执行迭代的每个线程调用一次。

第二个委托参数为循环体定义了委托。在示例中，该参数的类型是 Func<int, ParallelLoopState, string , string>。其中第一个参数是循环迭代，第二个参数 ParallelLoopState 允许停止循环，如前所述。循环体方法通过第 3 个参数接收从 init 方法返回的值，循环体方法还需要返回一个值，其类型是用泛型 For 参数定义的。

For()方法的最后一个参数指定一个委托 Action<TLocal>；在该示例中，接收一个字符串。这个方法仅对于每个线程调用一次，这是一个线程退出方法(代码文件 ParallelSamples/Program.cs)。

```
public static void ParallelForWithInit()
{
  Parallel.For<string>(0, 10, () =>
  {
    // invoked once for each thread
    Log($"init thread");
    return $"t{Thread.CurrentThread.ManagedThreadId}";
  },
  (i, pls, str1) =>
  {
    // invoked for each member
    Log($"body i {i} str1 {str1}");
    Task.Delay(10).Wait();
    return $"i {i}";
  },
  (str1) =>
  {
    // final action on each thread
    Log($"finally {str1}");
  });
}
```

运行一次这个程序的结果如下：

```
init thread task: 7, thread: 6
init thread task: 6, thread: 5
body i: 4 str1: t6 task: 7, thread: 6
body i: 2 str1: t5 task: 6, thread: 5
init thread task: 5, thread: 1
body i: 0 str1: t1 task: 5, thread: 1
init thread task: 9, thread: 8
body i: 8 str1: t8 task: 9, thread: 8
init thread task: 8, thread: 7
body i: 6 str1: t7 task: 8, thread: 7
body i: 1 str1: i 0 task: 5, thread: 1
finally i 2 task: 6, thread: 5
init thread task: 16, thread: 5
finally i 8 task: 9, thread: 8
init thread task: 17, thread: 8
body i: 9 str1: t8 task: 17, thread: 8
finally i 6 task: 8, thread: 7
init thread task: 18, thread: 7
body i: 7 str1: t7 task: 18, thread: 7
finally i 4 task: 7, thread: 6
init thread task: 15, thread: 10
body i: 3 str1: t10 task: 15, thread: 10
body i: 5 str1: t5 task: 16, thread: 5
finally i 1 task: 5, thread: 1
finally i 5 task: 16, thread: 5
finally i 3 task: 15, thread: 10
finally i 7 task: 18, thread: 7
finally i 9 task: 17, thread: 8
```

输出显示，为每个线程只调用一次 init()方法；循环体从初始化中接收第一个字符串，并用相同的线程将这个字符串传递到下一个迭代体。最后，为每个线程调用一次最后一个动作，从每个体中接收最后的结果。

通过这个功能，这个方法完美地累加了大量数据集合的结果。

## 21.2.4 使用 Parallel.ForEach()方法循环

Parallel.ForEach()方法遍历实现了 IEnumerable 的集合，其方式类似于 foreach 语句，但以异步方式遍历。这里也没有确定遍历顺序(代码文件 ParallelSamples/Program.cs)。

```
public static void ParallelForEach()
{
  string[] data = {"zero", "one", "two", "three", "four", "five",
  "six", "seven", "eight", "nine", "ten", "eleven", "twelve"};
  ParallelLoopResult result =
    Parallel.ForEach<string>(data, s =>
    {
      Console.WriteLine(s);
    });
}
```

如果需要中断循环，就可以使用 ForEach()方法的重载版本和 ParallelLoopState 参数。其方式与前面的 For()方法相同。ForEach()方法的一个重载版本也可以用于访问索引器，从而获得迭代次数，如下所示：

```
Parallel.ForEach<string>(data, (s, pls, l) =>
{
  Console.WriteLine($"{s} {l}");
});
```

### 21.2.5 通过 Parallel.Invoke()方法调用多个方法

如果多个任务将并行运行，就可以使用 Parallel.Invoke()方法，它提供了任务并行性模式。Parallel.Invoke()方法允许传递一个 Action 委托的数组，在其中可以指定将运行的方法。示例代码传递了要并行调用的 Foo()和 Bar()方法(代码文件 ParallelSamples/Program.cs)：

```
public static void ParallelInvoke()
{
  Parallel.Invoke(Foo, Bar);
}

public static void Foo() =>
  Console.WriteLine("foo");

public static void Bar() =>
  Console.WriteLine("bar");
```

Parallel 类使用起来十分方便，而且既可以用于任务，又可以用于数据并行性。如果需要更细致的控制，并且不想等到 Parallel 类结束后再开始动作，就可以使用 Task 类。当然，结合使用 Task 类和 Parallel 类也是可以的。

## 21.3 任务

为了更好地控制并行操作，可以使用 System.Threading.Tasks 名称空间中的 Task 类。任务表示将完成的某个工作单元。这个工作单元可以在单独的线程中运行，也可以以同步方式启动一个任务，这需要等待主调线程。使用任务不仅可以获得一个抽象层，还可以对底层线程进行很多控制。

在安排需要完成的工作时，任务提供了非常大的灵活性。例如，可以定义连续的工作——在一个任务完成后该执行什么工作。这可以根据任务成功与否来区分。另外，还可以在层次结构中安排任务。例如，父任务可以创建新的子任务。这可以创建一种依赖关系，这样，取消父任务，也会取消其子任务。

### 21.3.1 启动任务

要启动任务，可以使用 TaskFactory 类或 Task 类的构造函数和 Start()方法。Task 类的构造函数在创建任务上提供的灵活性较大。

TaskSamples 的示例代码使用如下名称空间：

**名称空间**

System

System.Linq

System.Threading

System.Threading.Tasks

在启动任务时，会创建 Task 类的一个实例，利用 Action 或 Action<object>委托(不带参数或带一个 object 参数)，可以指定将运行的代码。下面定义的方法 TaskMethod 带一个参数。在实现代码中，调用 Log 方法，把任务的 ID 和线程的 ID 写入控制台中，并且如果线程来自一个线程池，或者线程是一个后台线程，也要写入相关信息。把多条消息写入控制台的操作是使用 lock 关键字和 s_logLock 同步对象进行同步的。这样，就可以并行调用 Log，而且多次写入控制台的操作也不会彼此交叉。否则，title 可能由一个任务写入，而线程信息由另

一个任务写入(代码文件 TaskSamples/Program.cs)：

```
public static void TaskMethod(object o)
{
  Log(o?.ToString());
}

private static object s_logLock = new object();
public static void Log(string title)
{
  lock (s_logLock)
  {
    Console.WriteLine(title);
    Console.WriteLine($"Task id: {Task.CurrentId?.ToString() ?? "no task"}, " +
      $"thread: {Thread.CurrentThread.ManagedThreadId}");
    Console.WriteLine($"is pooled thread: " +
      $"{Thread.CurrentThread.IsThreadPoolThread}");
    Console.WriteLine($"is background thread: " +
      $"{Thread.CurrentThread.IsBackground}");
    Console.WriteLine();
  }
}
```

接下来的几小节描述了启动新任务的不同方法。

### 1. 使用线程池的任务

在本节中，可以看到启动使用了线程池中线程的任务的不同方式。线程池提供了一个后台线程的池。线程池独自管理线程，根据需要增加或减少线程池中的线程数。线程池中的线程用于实现一些操作，之后仍然返回线程池中。

创建任务的第一种方式是使用实例化的 TaskFactory 类，在其中把 TaskMethod 方法传递给 StartNew 方法，就会立即启动任务。第二种方式是使用 Task 类的静态属性 Factory 来访问 TaskFactory，以及调用 StartNew()方法。它与第一种方式很类似，也使用了工厂，但是对工厂创建的控制则没有那么全面。第三种方式是使用 Task 类的构造函数。实例化 Task 对象时，任务不会立即运行，而是指定 Created 状态。接着调用 Task 类的 Start()方法，来启动任务。第四种方式调用 Task 类的 Run 方法，立即启动任务。Run 方法没有可以传递 Action<object>委托的重载版本，但是通过传递 Action 类型的 lambda 表达式并在其实现中使用参数，可以模拟这种行为(代码文件 TaskSamples/Program.cs)。

```
public void TasksUsingThreadPool()
{
  var tf = new TaskFactory();
  Task t1 = tf.StartNew(TaskMethod, "using a task factory");
  Task t2 = Task.Factory.StartNew(TaskMethod, "factory via a task");
  var t3 = new Task(TaskMethod, "using a task constructor and Start");
  t3.Start();
  Task t4 = Task.Run(() => TaskMethod("using the Run method"));
}
```

这些版本返回的输出如下所示。它们都创建一个新任务，并使用线程池中的一个线程：

```
using a task factory
Task id: 1, thread: 4
is pooled thread: True
is background thread: True

factory via a task
Task id: 2, thread: 3
is pooled thread: True
is background thread: True

using a task constructor and Start
Task id: 3, thread: 5
is pooled thread: True
is background thread: True

using the Run method
Task id: 4, thread: 6
is pooled thread: True
is background thread: True
```

使用 Task 构造函数和 TaskFactory 的 StartNew()方法时，可以传递 TaskCreationOptions 枚举中的值。利用这个创建选项，可以改变任务的行为，如接下来的小节所示。

#### 2. 同步任务

任务不一定要使用线程池中的线程，也可以使用其他线程。任务也可以同步运行，以相同的线程作为主调线程。下面的代码段使用了 Task 类的 RunSynchronously()方法(代码文件 TaskSamples/Program.cs)：

```
private static void RunSynchronousTask()
{
  TaskMethod("just the main thread");
  var t1 = new Task(TaskMethod, "run sync");
  t1.RunSynchronously();
}
```

这里，TaskMethod()方法首先在主线程上直接调用，然后在新创建的 Task 上调用。从如下所示的控制台输出可以看到，主线程没有任务 ID，也不是线程池中的线程。调用 RunSynchronously()方法时，会使用相同的线程作为主调线程，但是如果以前没有创建任务，就会创建一个任务：

```
just the main thread
Task id: no task, thread: 2
is pooled thread: False
is background thread: False

run sync
Task id: 1, thread: 2
is pooled thread: False
is background thread: False
```

#### 3. 使用单独线程的任务

如果任务的代码将长时间运行，就应该使用 TaskCreationOptions.LongRunning 告诉任务调度器创建一个新线程，而不是使用线程池中的线程。此时，线程可以不由线程池管理。当线程来自线程池时，任务调度器可以决定等待已经运行的任务完成，然后使用这个线程，而不是在线程池中创建一个新线程。对于长时间运行的线程，任务调度器会立即知道等待它们完成没有意义。下面的代码片段创建了一个长时间运行的任务(代码文件 TaskSamples/Program.cs)：

```
private static void LongRunningTask()
{
  var t1 = new Task(TaskMethod, "long running",
    TaskCreationOptions.LongRunning);
  t1.Start();
}
```

实际上，使用 TaskCreationOptions.LongRunning 选项时，不会使用线程池中的线程，而是创建一个新线程：

```
long running
Task id: 5, thread: 7
is pooled thread: False
is background thread: True
```

### 21.3.2 Future——任务的结果

当任务结束时，它可以把一些有用的状态信息写到共享对象中。这个共享对象必须是线程安全的。另一个选项是使用返回某个结果的任务。这种任务也称为 future，因为它在将来返回一个结果。早期版本的 Task Parallel Library(TPL)的类名也称为 Future，现在它是 Task 类的一个泛型版本。使用这个类时，可以定义任务返回的结果的类型。

由任务调用返回结果的方法可以声明为任何返回类型。下面的示例方法 TaskWithResult()利用一个元组返回两个 int 值。该方法的输入可以是 void 或 object 类型，如下所示(代码文件 TaskSamples/Program.cs)：

```
public static (int Result, int Remainder) TaskWithResult(object division)
{
  (int x, int y) = ((int x, int y))division;
  int result = x / y;
```

```
    int remainder = x % y;
    Console.WriteLine("task creates a result...");
    return (result, remainder);
  }
```

**注意：**
元组允许把多个值组合为一个，参见第 13 章。

当定义一个调用 TaskWithResult()方法的任务时，要使用泛型类 Task<TResult>。泛型参数定义了返回类型。通过构造函数，把这个方法传递给 Func 委托，第二个参数定义了输入值。因为这个任务在 object 参数中需要两个输入值，所以还创建了一个元组。接着启动该任务。Task 实例 t1 块的 Result 属性被禁用，并一直等到该任务完成。任务完成后，Result 属性包含任务的结果。

```
public static void TaskWithResultDemo()
{
  var t1 = new Task<(int Result, int Remainder)>(TaskWithResult, (8, 3));
  t1.Start();
  Console.WriteLine(t1.Result);
  t1.Wait();
  Console.WriteLine($"result from task: {t1.Result.Result} " +
    $"{t1.Result.Remainder}");
}
```

### 21.3.3　连续的任务

通过任务，可以指定在任务完成后，应开始运行另一个特定任务，例如，一个新任务使用前一个任务的结果，如果前一个任务失败了，这个任务就应执行一些清理工作。

任务处理程序或者不带参数，或者带一个对象参数，而连续处理程序有一个 Task 类型的参数，这里可以访问起始任务的相关信息(代码文件 TaskSamples/Program.cs)：

```
private static void DoOnFirst()
{
  Console.WriteLine($"doing some task {Task.CurrentId}");
  Task.Delay(3000).Wait();
}

private static void DoOnSecond(Task t)
{
  Console.WriteLine($"task {t.Id} finished");
  Console.WriteLine($"this task id {Task.CurrentId}");
  Console.WriteLine("do some cleanup");
  Task.Delay(3000).Wait();
}
```

连续任务通过在任务上调用 ContinueWith()方法来定义。也可以使用 TaskFactory 类来定义。t1.OnContinueWith(DoOnSecond)方法表示，调用 DoOnSecond()方法的新任务应在任务 t1 结束时立即启动。在一个任务结束时，可以启动多个任务，连续任务也可以有另一个连续任务，如下面的例子所示(代码文件 TaskSamples/Program.cs)：

```
public static void ContinuationTasks()
{
  Task t1 = new Task(DoOnFirst);
  Task t2 = t1.ContinueWith(DoOnSecond);
  Task t3 = t1.ContinueWith(DoOnSecond);
  Task t4 = t2.ContinueWith(DoOnSecond);
  t1.Start();
}
```

无论前一个任务是如何结束的，前面的连续任务总是在前一个任务结束时启动。使用 TaskContinuationOptions 枚举中的值可以指定，连续任务只有在起始任务成功(或失败)结束时启动。一些可能的值是 OnlyOnFaulted、NotOnFaulted、OnlyOnCanceled、NotOnCanceled 以及 OnlyOnRanToCompletion。

```
Task t5 = t1.ContinueWith(DoOnError, TaskContinuationOptions.OnlyOnFaulted);
```

> **注意：**
> 使用第 15 章介绍过的 await 关键字时，编译器生成的代码会使用连续任务。

### 21.3.4 任务层次结构

利用任务连续性，可以在一个任务结束后启动另一个任务。任务也可以构成一个层次结构。一个任务启动一个新任务时，就启动了一个父/子层次结构。

下面的代码段在父任务内部新建一个任务对象并启动任务。创建子任务的代码与创建父任务的代码相同，唯一的区别是这个任务从另一个任务内部创建(代码文件 TaskSamples/Program.cs)：

```
public static void ParentAndChild()
{
  var parent = new Task(ParentTask);
  parent.Start();
  Task.Delay(2000).Wait();
  Console.WriteLine(parent.Status);
  Task.Delay(4000).Wait();
  Console.WriteLine(parent.Status);
}

private static void ParentTask()
{
  Console.WriteLine($"task id {Task.CurrentId}");
  var child = new Task(ChildTask);
  child.Start();
  Task.Delay(1000).Wait();
  Console.WriteLine("parent started child");
}

private static void ChildTask()
{
  Console.WriteLine("child");
  Task.Delay(5000).Wait();
  Console.WriteLine("child finished");
}
```

如果父任务在子任务之前结束，父任务的状态就显示为 WaitingForChildrenToComplete。所有的子任务也结束时，父任务的状态就变成 RanToCompletion。当然，如果父任务用 TaskCreationOptionDetachedFromParent 创建一个任务时，这就无效。

取消父任务，也会取消子任务。接下来就讨论取消架构。

### 21.3.5 从方法中返回任务

返回任务和结果的方法声明为返回 Task< T >，例如，方法返回一个任务和字符串集合：

```
public Task<IEnumerable<string>> TaskMethodAsync()
{
}
```

创建访问网络或数据的方法通常是异步的，这样，就可以使用任务特性来处理结果(例如使用 async 关键字，参见第 15 章)。如果有同步路径，或者需要实现一个用同步代码定义的接口，就不需要为了结果的值创建一个任务。Task 类使用方法 FromResult()创建已完成任务的结果，该任务用状态 RanToCompletion 表示完成：

```
return Task.FromResult<IEnumerable<string>>(
  new List<string>() { "one", "two" });
```

### 21.3.6 等待任务

也许读者学习过 Task 类的 WhenAll()和 WaitAll()方法，想知道它们之间的区别。这两个方法都等待传递给它们的所有任务的完成。WaitAll()方法阻塞调用任务，直到等待的所有任务完成为止。WhenAll()方法返回一个任务，从而允许使用 async 关键字等待结果，它不会阻塞等待的任务。

在等待的所有任务都完成后，WhenAll()和 WaitAll()方法才完成，而使用 WhenAny()和 WaitAny()方法，可以

等待任务列表中的一个任务完成。类似于 WhenAll()和 WaitAll()方法，WaitAny()方法会阻塞任务的调用，而 WhenAny()返回可以等待的任务。

前面几个示例已经使用了 Task.Delay()方法。可以指定这个方法返回的任务完成前要等待的毫秒数。

如果将释放 CPU，从而允许其他任务运行，就可以调用 Task.Yield()方法。该方法释放 CPU，让其他任务运行。如果没有其他的任务等待运行，调用 Task.Yield 的任务就立即继续执行。否则，需要等到再次调度 CPU，以调用任务。

### ValueTask

如果方法有时是异步运行的，但并不总是这样，Task 类可能有一些不需要的开销。.NET 现在提供了 ValueTask，它是一个结构，相对于 Task 类，这样 ValueTask 就没有堆中对象的开销了。通常调用异步方法，例如对 API 服务器或数据库进行调用，与需要完成工作的时间相比，Task 类型的开销可以忽略。然而，在某些情况下，不能忽略开销，例如，方法被调用数千次时，很少真正需要通过网络进行调用。在这个场景中，ValueTask 变得非常方便。

下面看一个例子。方法 GetTheRealData 模拟通常需要很长时间的方法，在网络或数据库上访问数据。在这里，使用 Enumerable 类生成示例数据。随着时间的推移，检索数据，结果以元组的形式返回。该方法返回我们常用的 Task(代码文件 ValueTaskSample/Program.cs)：

```
public static Task<(IEnumerable<string> data, DateTime retrievedTime)>
  GetTheRealData() =>
    Task.FromResult(
      (Enumerable.Range(0, 10)
        .Select(x => $"item {x}").AsEnumerable(), DateTime.Now));
```

有趣的部分现在在方法 GetSomeData 中。这个方法声明为返回一个 ValueTask。在实现中，如果缓存的数据不超过 5 秒，则首先进行检查。如果缓存的数据没有变旧，就直接返回缓存的数据，并传递给 ValueTask 构造函数。这并不需要后台线程；数据可以直接返回。如果缓存较老，则调用 GetTheRealData 方法。这个方法需要一个真正的任务，并且可能会出现一些延迟(代码文件 ValueTaskSample/Program.cs)：

```
private static DateTime _retrieved;
private static IEnumerable<string> _cachedData;
public static async ValueTask<IEnumerable<string>> GetSomeDataAsync()
{
  if (_retrieved >= DateTime.Now.AddSeconds(-5))
  {
    Console.WriteLine("data from the cache");
    return await new ValueTask<IEnumerable<string>>(_cachedData);
  }

  Console.WriteLine("data from the service");
  (_cachedData, _retrieved) = await GetTheRealData();
  return _cachedData;
}
```

**注意：**

ValueTask 的构造函数要为返回的数据接受类型 TResult 或 Task<TResult>，来提供从异步运行的方法中返回的 Task。

Main()方法包括一个循环，在每次迭代之后，多次调用 GetSomeDataAsync()方法(代码文件 ValueTaskSample/Program.cs)：

```
static async Task Main(string[] args)
{
  for (int i = 0; i < 20; i++)
  {
    IEnumerable<string> data = await GetSomeDataAsync();
    await Task.Delay(1000);
  }
  Console.ReadLine();
}
```

运行应用程序时，可以看到数据从缓存中返回，并且在缓存失效之后，在再次使用缓存之前访问服务。

```
data from the service
data from the cache
data from the cache
data from the cache
data from the cache
data from the service
data from the cache
data from the cache
data from the cache
data from the cache
data from the service
data from the cache
...
```

**注意：**
与 ValueTask 相比，可能不能忽略任务的开销。但是，在框架中拥有这个核心功能，可以在未来的 C#版本中实现异步流或异步操作符等未来特性。

# 21.4 取消架构

.NET 包含一个取消架构，允许以标准方式取消长时间运行的任务。每个阻塞调用都应支持这种机制。当然目前并不是所有阻塞调用都实现了这个新技术，但越来越多的阻塞调用都支持它。已经提供了这种机制的技术有任务、并发集合类、并行 LINQ 和几种同步机制。

取消架构基于协作行为，它不是强制的。长时间运行的任务会检查它是否被取消，并相应地返回控制权。

支持取消的方法接受一个 CancellationToken 参数。这个类定义了 IsCancellationRequested 属性，其中长时间运行的操作可以检查它是否应终止。长时间运行的操作检查取消的其他方式有：取消标记时，使用标记的 WaitHandle 属性，或者使用 Register()方法。Register()方法接受 Action 和 ICancelableOperation 类型的参数。Action 委托引用的方法在取消标记时调用。这类似于 ICancelableOperation，其中实现这个接口的对象的 Cancel()方法在执行取消操作时调用。

CancellationSamples 的示例代码使用如下名称空间：

**名称空间**

System

System.Threading

System.Threading.Tasks

## 21.4.1 Parallel.For()方法的取消

本节以一个使用 Parallel.For()方法的简单例子开始。Parallel 类提供了 For()方法的重载版本，在重载版本中，可以传递 ParallelOptions 类型的参数。使用 ParallelOptions 类型，可以传递一个 CancellationToken 参数。CancellationToken 参数通过创建 CancellationTokenSource 来生成。由于 CancellationTokenSource 实现了 ICancelableOperation 接口，因此可以用 CancellationToken 注册，并允许使用 Cancel()方法取消操作。本例没有直接调用 Cancel()方法，而是使用了方法 CancelAfter()，在 500 毫秒后取消标记。

在 For()循环的实现代码内部，Parallel 类验证 CancellationToken 的结果，并取消操作。一旦取消操作，For()方法就抛出一个 OperationCanceledException 类型的异常，这是本例捕获的异常。使用 CancellationToken 可以注册取消操作时的信息。为此，需要调用 Register()方法，并传递一个在取消操作时调用的委托(代码文件 CancellationSamples/Program.cs)。

```
public static void CancelParallelFor()
{
  var cts = new CancellationTokenSource();
  cts.Token.Register(() => Console.WriteLine("*** token cancelled"));
  // send a cancel after 500 ms
  cts.CancelAfter(500);
  try
```

```
    {
      ParallelLoopResult result =
        Parallel.For(0, 100, new ParallelOptions
        {
          CancellationToken = cts.Token,
        },
        x =>
        {
          Console.WriteLine($"loop {x} started");
          int sum = 0;
          for (int i = 0; i < 100; i++)
          {
            Task.Delay(2).Wait();
            sum += i;
          }
          Console.WriteLine($"loop {x} finished");
        });
    }
    catch (OperationCanceledException ex)
    {
      Console.WriteLine(ex.Message);
    }
  }
```

运行应用程序，会得到类似如下的结果，第 0、50、25、75 和 1 次迭代都启动了。这在一个有 4 个内核 CPU 的系统上运行。通过取消操作，所有其他的迭代操作都在启动之前就取消了。启动的迭代操作允许完成，因为取消操作总是以协作方式进行，以避免在取消迭代操作的中间泄漏资源。

```
loop 0 started
loop 50 started
loop 25 started
loop 75 started
loop 1 started
*** token cancelled
loop 75 finished
loop 50 finished
loop 1 finished
loop 0 finished
loop 25 finished
The operation was canceled.
```

## 21.4.2　任务的取消

同样的取消模式也可用于任务。首先，新建一个 CancellationTokenSource。如果仅需要一个取消标记，就可以通过访问 Task.Factory.CancellationToken 以使用默认的取消标记。接着，与前面的代码类似，在 500 毫秒后取消任务。在循环中执行主要工作的任务通过 TaskFactory 对象接受取消标记。在构造函数中，把取消标记赋予 TaskFactory。这个取消标记由任务用于检查 CancellationToken 的 IsCancellationRequested 属性，以确定是否请求了取消(代码文件 CancellationSamples/Program.cs)。

```
public void CancelTask()
{
  var cts = new CancellationTokenSource();
  cts.Token.Register(() => Console.WriteLine("*** task cancelled"));
  // send a cancel after 500 ms
  cts.CancelAfter(500);
  Task t1 = Task.Run(() =>
  {
    Console.WriteLine("in task");
    for (int i = 0; i < 20; i++)
    {
      Task.Delay(100).Wait();
      CancellationToken token = cts.Token;
      if (token.IsCancellationRequested)
      {
        Console.WriteLine("cancelling was requested, " +
          "cancelling from within the task");
        token.ThrowIfCancellationRequested();
        break;
      }
      Console.WriteLine("in loop");
    }
    Console.WriteLine("task finished without cancellation");
```

```
  }, cts.Token);

  try
  {
    t1.Wait();
  }
  catch (AggregateException ex)
  {
    Console.WriteLine($"exception: {ex.GetType().Name}, {ex.Message}");
    foreach (var innerException in ex.InnerExceptions)
    {
      Console.WriteLine($"inner exception: {ex.InnerException.GetType()}," +
        $"{ex.InnerException.Message}");
    }
  }
}
```

运行应用程序，可以看到任务启动了，运行了几个循环，并获得了取消请求。之后取消任务，并抛出 TaskCanceledException 异常，它是从方法调用 ThrowIfCancellationRequested()中启动的。调用者等待任务时，会捕获 AggregateException 异常，它包含内部异常 TaskCanceledException。例如，如果在一个也被取消的任务中运行 Parallel.For()方法，这就可以用于取消的层次结构。任务的最终状态是 Canceled。

```
in task
in loop
in loop
in loop
in loop
*** task cancelled
cancelling was requested, cancelling from within the task
exception: AggregateException, One or more errors occurred.
inner exception: TaskCanceledException, A task was canceled.
```

# 21.5 数据流

Parallel 类、Task 类和 Parallel LINQ 为数据并行性提供了很多帮助。但是，这些类不能直接支持数据流的处理，以及并行转换数据。此时，需要使用 Task Parallel Library Data Flow(TPL Data Flow)。

数据流示例的代码使用了如下名称空间：

**名称空间**

System

System.IO

System.Threading

System.Threading.Tasks

System.Threading.Tasks.DataFlow

## 21.5.1 使用动作块

TPL Data Flow 的核心是数据块，这些数据块作为提供数据的源或者接收数据的目标，或者同时作为源和目标。下面看一个简单的示例，其中用一个数据块来接收一些数据并把数据写入控制台。下面的代码段定义了一个 ActionBlock，它接收一个字符串，并把字符串中的信息写入控制台。Main()方法在一个 while 循环中读取用户输入，然后调用 Post()方法把读入的所有字符串写入 ActionBlock，Post()方法把一项传递给 ActionBlock。ActionBlock 异步处理消息，把信息写入控制台(代码文件 SimpleDataFlowSample/Program.cs)：

```
static void Main()
{
  var processInput = new ActionBlock<string>(s =>
  {
    Console.WriteLine($"user input: {s}");
  });
  bool exit = false;
  while (!exit)
  {
    string input = ReadLine();
```

```
        if (string.Compare(input, "exit", ignoreCase: true) == 0)
        {
          exit = true;
        }
        else
        {
          processInput.Post(input);
        }
      }
    }
```

## 21.5.2 源和目标数据块

以前示例中分配给 ActionBlock 的方法执行时，ActionBlock 会使用一个任务来并行执行。通过检查任务和线程标识符，并把它们写入控制台可以验证这一点。每个块都实现了 IDataflowBlock 接口，该接口包含了返回一个 Task 的属性 Completion，以及 Complete()和 Fault()方法。调用 Complete()方法后，块不再接受任何输入，也不再产生任何输出。调用 Fault()方法则把块放入失败状态。

如前所述，块既可以是源，也可以是目标，还可以同时是源和目标。在示例中，ActionBlock 是一个目标块，所以实现了 ITargetBlock 接口。ITargetBlock 派生自 IDataflowBlock，除了提供 IDataBlock 接口的成员以外，还定义了 OfferMessage()方法。OfferMessage()发送一条由块处理的消息。Post 是比 OfferMessage 更方便的一个方法，它实现为 ITargetBlock 接口的扩展方法。示例应用程序中也使用了 Post()方法。

ISourceBlock 接口由作为数据源的块实现。除了 IDataBlock 接口的成员以外，ISourceBlock 还提供了链接到目标块以及处理消息的方法。

BufferBlock 同时作为数据源和数据目标，它实现了 ISourceBlock 和 ITargetBlock。在下一个示例中，就使用这个 BufferBlock 收发消息(代码文件 SimpleDataFlowSample/Program.cs)：

Producer()方法从控制台读取字符串，并通过调用 Post()方法把字符串写到 BufferBlock 中：

```
public static void Producer()
{
  bool exit = false;
  while (!exit)
  {
  string input = ReadLine();
  if (string.Compare(input, "exit", ignoreCase: true) == 0)
  {
    exit = true;
  }
  else
  {
    s_buffer.Post(input);
  }
}
```

Consumer()方法在一个循环中调用 ReceiveAsync()方法来接收 BufferBlock 中的数据。ReceiveAsync 是 ISourceBlock 接口的一个扩展方法：

```
public static async Task ConsumerAsync()
{
  while (true)
  {
    string data = await s_buffer.ReceiveAsync();
    Console.WriteLine($"user input: {data}");
  }
}
```

现在，只需要启动消息的产生者和使用者。在 Main()方法中通过两个独立的任务完成启动操作：

```
static void Main()
{
  Task t1 = Task.Run(() => Producer());
  Task t2 = Task.Run(async () => await ConsumerAsync());
  Task.WaitAll(t1, t2);
}
```

运行应用程序时，产生者从控制台读取数据，使用者接收数据并把它们写入控制台。

### 21.5.3 连接块

本节将连接多个块，创建一个管道。首先，创建由块使用的 3 个方法。GetFileNames()方法接收一个目录路径作为参数，得到以.cs 为扩展名的文件名(代码文件 DataFlowSample/Program.cs)：

```
public static IEnumerable<string> GetFileNames(string path)
{
  foreach (var fileName in Directory.EnumerateFiles(path, "*.cs"))
  {
    yield return fileName;
  }
}
```

LoadLines()方法以一个文件名列表作为参数，得到文件中的每一行：

```
public static IEnumerable<string> LoadLines(IEnumerable<string> fileNames)
{
  foreach (var fileName in fileNames)
  {
    using (FileStream stream = File.OpenRead(fileName))
    {
      var reader = new StreamReader(stream);
      string line = null;
      while ((line = reader.ReadLine()) != null)
      {
        //WriteLine($"LoadLines {line}");
        yield return line;
      }
    }
  }
}
```

GetWords()方法接收一个 lines 集合作为参数，将其逐行分割，从而得到并返回一个单词列表：

```
public static IEnumerable<string> GetWords(IEnumerable<string> lines)
{
  foreach (var line in lines)
  {
    string[] words = line.Split(' ', ';', '(', ')', '{', '}', '.', ',');
    foreach (var word in words)
    {
      if (!string.IsNullOrEmpty(word))
        yield return word;
    }
  }
}
```

为了创建管道，SetupPipeline()方法创建了 3 个 TransformBlock 对象。TransformBlock 是一个源和目标块，通过使用委托来转换源。第一个 TransformBlock 被声明为将一个字符串转换为 IEnumerable<string>。这种转换是通过 GetFileNames()方法完成的，GetFileNames()方法在传递给第一个块的构造函数的 lambda 表达式中调用。类似地，接下来的两个 TransformBlock 对象用于调用 LoadLines()和 GetWords()方法：

```
public static ITargetBlock<string> SetupPipeline()
{
  var fileNamesForPath = new TransformBlock<string, IEnumerable<string>>(
    path => GetFileNames(path));

  var lines = new TransformBlock<IEnumerable<string>, IEnumerable<string>>(
    fileNames => LoadLines(fileNames));

  var words = new TransformBlock<IEnumerable<string>, IEnumerable<string>>(
    lines2 => GetWords(lines2));
```

定义的最后一个块是 ActionBlock。这个块只是一个用于接收数据的目标块，前面已经用过：

```
var display = new ActionBlock<IEnumerable<string>>(
  coll =>
  {
    foreach (var s in coll)
    {
      Console.WriteLine(s);
    }
  });
```

最后，将这些块彼此连接起来。fileNamesForPath 被链接到 lines 块，其结果被传递给 lines 块。lines 块链接到 words 块，words 块链接到 display 块。最后，返回用于启动管道的块：

```
  fileNamesForPath.LinkTo(lines);
  lines.LinkTo(words);
  words.LinkTo(display);
  return fileNamesForPath;
}
```

现在，Main()方法只需要启动管道。调用 Post()方法传递目录时，管道就会启动，并最终将单词从 C#源代码写入控制台。这里可以发出多个启动管道的请求，传递多个目录，并行执行这些任务：

```
static void Main()
{
  var target = SetupPipeline();
  target.Post(".");
  Console.ReadLine();
}
```

通过对 TPL Data Flow 库的简单介绍，可以看到这种技术的主要用法。该库还提供了其他许多功能，例如以不同方式处理数据的不同块。BroadcastBlock 允许向多个目标传递输入源(例如将数据写入一个文件并显示该文件)，JoinBlock 将多个源连接到一个目标，BatchBlock 将输入作为数组进行批处理。使用 DataflowBlockOptions 选项可以配置块，例如一个任务中可以处理的最大项数，还可以向其传递取消标记来取消管道。使用链接技术，可以对消息进行筛选，只传递满足指定条件的消息。

## 21.6　Timer 类

使用计时器，可以重复调用方法。本节介绍两个计时器：System.Threading 名称空间中的 Timer 类和用于基于 XAML 应用程序的 DispatcherTimer。

使用 System.Threading.Timer 类，可以把要调用的方法作为构造函数的第一个参数传递。这个方法必须满足 TimeCallback 委托的要求，该委托定义一个 void 返回类型和一个 object 参数。通过构造函数的第二个参数，可以传递任意对象，用回调方法中的 object 参数接收对应的对象。例如，可以传递 Event 对象，向调用者发送信号。第 3 个参数指定第一次调用回调方法时的时间段。最后一个参数指定回调的重复时间间隔。如果计时器应只触发一次，就把第 4 个参数设置为值–1。

如果创建 Timer 对象后应改变时间间隔，就可以用 Change()方法传递新值(代码文件 TimerSample/Program.cs)：

```
private static void ThreadingTimer()
{
  void TimeAction(object o) =>
    Console.WriteLine($"System.Threading.Timer {DateTime.Now:T}");

  using (var t1 = new Timer(TimeAction, null,
    TimeSpan.FromSeconds(2), TimeSpan.FromSeconds(3)))
  {
    Task.Delay(15000).Wait();
  }
}
```

Windows.UI.Xaml 名称空间(用于 UWP 应用程序)中的 DispatcherTimer 是一个基于 XAML 的应用程序的计时器，其中的事件处理程序在 UI 线程中调用，因此可以直接访问用户界面元素。

演示 DispatcherTimer 的示例应用程序是一个 Windows 应用程序，显示了切换每一秒的时钟指针。下面的 XAML 代码定义的命令允许开始和停止时钟(代码文件 WinAppTimer/MainPage.xaml)：

```
<Page.TopAppBar>
  <CommandBar IsOpen="True">
    <AppBarButton Icon="Play" Click="{x:Bind OnTimer}" />
    <AppBarButton Icon="Stop" Click="{x:Bind OnStopTimer}" />
  </CommandBar>
</Page.TopAppBar>
```

时钟的指针使用形状 Line 定义。要旋转该指针，请使用 RotateTransform 元素：

```
<Canvas Width="300" Height="300">
  <Ellipse Width="10" Height="10" Fill="Red" Canvas.Left="145"
    Canvas.Top="145" />
  <Line Canvas.Left="150" Canvas.Top="150" Fill="Green" StrokeThickness="3"
    Stroke="Blue" X1="0" Y1="0" X2="120" Y2="0" >
    <Line.RenderTransform>
      <RotateTransform CenterX="0" CenterY="0" Angle="270" x:Name="rotate" />
    </Line.RenderTransform>
  </Line>
</Canvas>
```

**注意：**
XAML 形状参见第 35 章。

DispatcherTimer 对象在 MainPage 类中创建。在构造函数中，处理程序方法分配给 Tick 事件，Interval 指定为 1 秒。在 OnTimer 方法中启动计时器，该方法在用户单击 CommandBar 中的 Play 按钮时调用(代码文件 WinAppTimer/MainPage.xaml.cs)：

```
private DispatcherTimer _timer = new DispatcherTimer();
public MainPage()
{
  this.InitializeComponent();
  _timer.Tick += OnTick;
  _timer.Interval = TimeSpan.FromSeconds(1);
}

private void OnTimer()
{
  _timer.Start();
}

private void OnTick(object sender, object e)
{
  double newAngle = rotate.Angle + 6;
  if (newAngle >= 360) newAngle = 0;
  rotate.Angle = newAngle;
}

private void OnStopTimer()
{
  _timer.Stop();
}
```

运行应用程序，就会显示时钟，如图 21-1 所示。

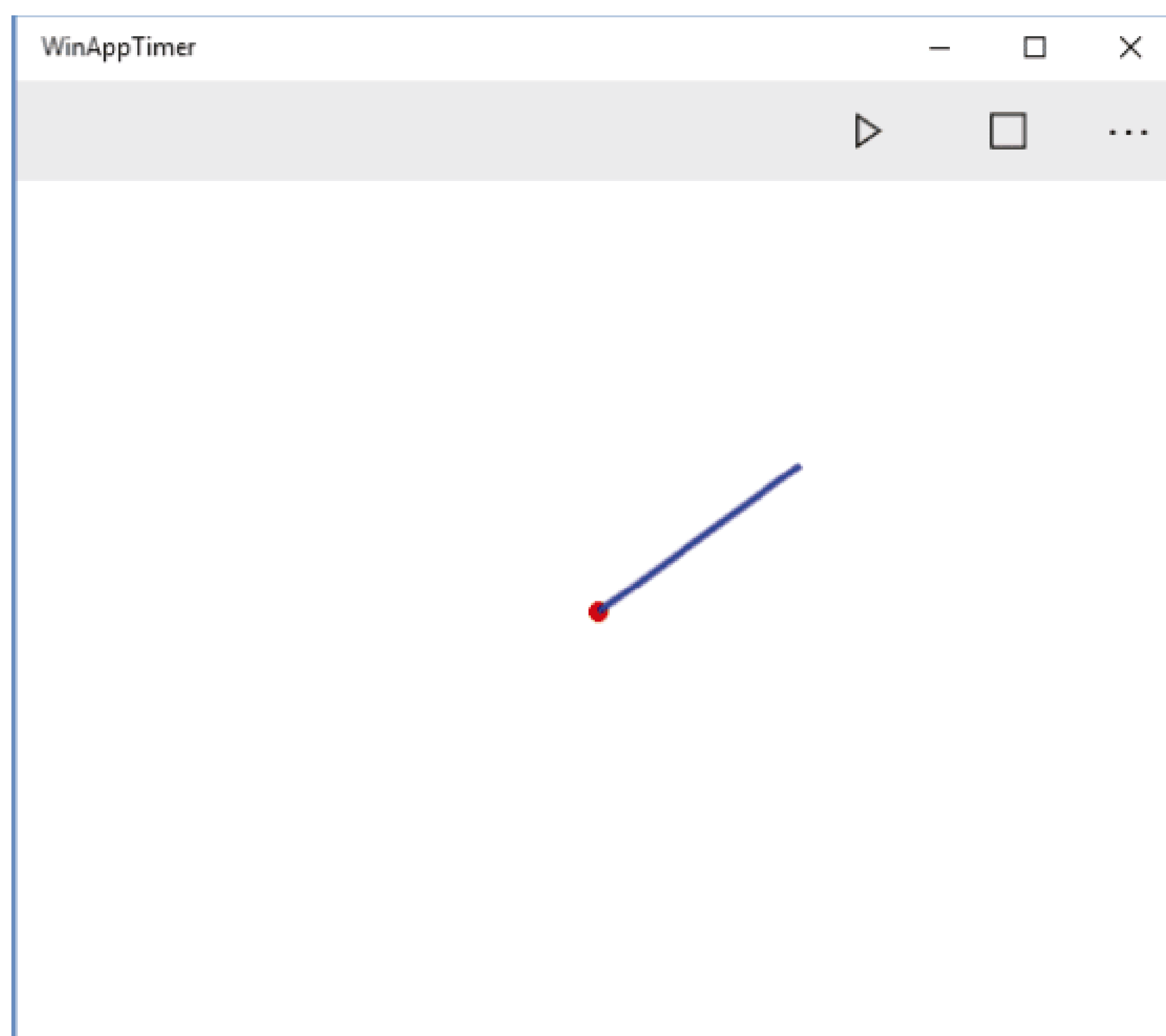



图 21-1

# 21.7 线程问题

用多个线程编程并不容易。在启动访问相同数据的多个线程时，会间歇性地遇到难以发现的问题。如果使用任务、并行 LINQ 或 Parallel 类，也会遇到这些问题。为了避免这些问题，必须特别注意同步问题和多个线程可能发生的其他问题。下面探讨与线程相关的问题：争用条件和死锁。

**注意：**
*在用这里显示的同步类型同步自定义集合类之前，还应该阅读第 11 章，了解线程安全的集合：并发集合。*

ThreadingIssues 示例的代码使用了如下名称空间：

System.Diagnostics

System.Threading

System.Threading.Tasks

static System.Console

可以使用命令行参数启动 ThreadingIssues 示例应用程序，来模拟争用条件或死锁。

## 21.7.1 争用条件

如果两个或多个线程访问相同的对象，并且对共享状态的访问没有同步，就会出现争用条件。为了说明争用条件，下面的例子定义一个 StateObject 类，它包含一个 int 字段和一个 ChangeState()方法。在 ChangeState()方法的实现代码中，验证状态变量是否包含 5。如果它包含，就递增其值。下一条语句是 Trace.Assert，它立刻验证 state 现在是包含 6。

在给包含 5 的变量递增了 1 后，可能认为该变量的值就是 6。但事实不一定是这样。例如，如果一个线程刚刚执行完 if (_state == 5)语句，它就被其他线程抢占，调度器运行另一个线程。第二个线程现在进入 if 体，因为 state 的值仍是 5，所以将它递增到 6。第一个线程现在再次被调度，在下一条语句中，state 递增到 7。这时就发生了争用条件，并显示断言消息(代码文件 ThreadingIssues/SampleTask.cs)。

```
public class StateObject
{
  private int _state = 5;
  public void ChangeState(int loop)
  {
    if (_state == 5)
    {
      _state++;
      if (_state != 6)
      {
        Console.WriteLine($"Race condition occurred after {loop} loops");
        Trace.Fail("race condition");
      }
    }
    _state = 5;
  }
}
```

下面通过给任务定义一个方法来验证这一点。SampleTask 类的 RaceCondition()方法将一个 StateObject 类作为其参数。在一个无限 while 循环中，调用 ChangeState()方法。变量 i 仅用于显示断言消息中的循环次数。

```
public class SampleTask
{
  public void RaceCondition(object o)
  {
    Trace.Assert(o is StateObject, "o must be of type StateObject");
    StateObject state = o as StateObject;
    int i = 0;
    while (true)
    {
      state.ChangeState(i++);
    }
  }
}
```

在程序的 Main()方法中，新建了一个 StateObject 对象，它由所有任务共享。通过使用传递给 Task 的 Run 方法的 lambda 表达式调用 RaceCondition 方法来创建 Task 对象。然后，主线程等待用户输入。但是，因为可能出现争用，所以程序很有可能在读取用户输入前就挂起：

```
public void RaceConditions()
{
  var state = new StateObject();
  for (int i = 0; i < 2; i++)
  {
    Task.Run(() => new SampleTask().RaceCondition(state));
  }
}
```

启动程序，就会出现争用条件。多久以后出现第一个争用条件要取决于系统以及将程序构建为发布版本还是调试版本。如果构建为发布版本，该问题的出现次数就会比较多，因为代码被优化了。如果系统中有多个 CPU 或使用双核/四核 CPU，其中多个线程可以同时运行，则该问题也会比单核 CPU 的出现次数多。在单核 CPU 中，因为线程调度是抢占式的，也会出现该问题，只是没有那么频繁。

在我的系统上运行程序时，显示在 85232 个循环后出现错误；在另一次运行程序时，显示在 70037 个循环后出现错误。多次启动应用程序，总是会得到不同的结果。

要避免该问题，可以锁定共享的对象。这可以在线程中完成：用下面的 lock 语句锁定在线程中共享的 state 变量。只有一个线程能在锁定块中处理共享的 state 对象。由于这个对象在所有的线程之间共享，因此，如果一个线程锁定了 state，另一个线程就必须等待该锁定的解除。一旦接受锁定，线程就拥有该锁定，直到该锁定块的末尾才解除锁定。如果改变 state 变量引用的对象的每个线程都使用一个锁定，就不会出现争用条件。

```
public class SampleTask
{
  public void RaceCondition(object o)
  {
    Trace.Assert(o is StateObject, "o must be of type StateObject");
    StateObject state = o as StateObject;
    int i = 0;
    while (true)
    {
      lock (state) // no race condition with this lock
      {
        state.ChangeState(i++);
      }
    }
  }
}
```

**注意：**
*在下载的示例代码中，需要取消锁定语句的注释，才能解决争用条件的问题。*

在使用共享对象时，除了进行锁定之外，还可以将共享对象设置为线程安全的对象。在下面的代码中，ChangeState()方法包含一条 lock 语句。由于不能锁定 state 变量本身(只有引用类型才能用于锁定)，因此定义一个 object 类型的变量 sync，将它用于 lock 语句。如果每次 state 的值更改时，都使用同一个同步对象来锁定，就不会出现争用条件。

```
public class StateObject
{
  private int _state = 5;
  private _object sync = new object();
  public void ChangeState(int loop)
  {
    lock (_sync)
    {
      if (_state == 5)
      {
        _state++;
        if (_state != 6)
        {
          Console.WriteLine($"Race condition occured after {loop} loops");
          Trace.Fail($"race condition at {loop}");
        }
```

```
        }
        _state = 5;
      }
    }
  }
```

## 21.7.2　死锁

过多的锁定也会有麻烦。在死锁中，至少有两个线程被挂起，并等待对方解除锁定。由于两个线程都在等待对方，就出现了死锁，线程将无限等待下去。

为了说明死锁，下面实例化 StateObject 类型的两个对象，并把它们传递给 SampleTask 类的构造函数。创建两个任务，其中一个任务运行 Deadlock1()方法，另一个任务运行 Deadlock2()方法(代码文件 ThreadingIssues/Program.cs)：

```
var state1 = new StateObject();
var state2 = new StateObject();
new Task(new SampleTask(state1, state2).Deadlock1).Start();
new Task(new SampleTask(state1, state2).Deadlock2).Start();
```

Deadlock1()和 Deadlock2()方法现在改变两个对象 s1 和 s2 的状态，所以生成了两个锁。Deadlock1()方法先锁定 s1，接着锁定 s2。Deadlock2()方法先锁定 s2，再锁定 s1。现在，有可能 Deadlock1()方法中 s1 的锁定会被解除。接着，出现一次线程切换，Deadlock2()方法开始运行，并锁定 s2。第二个线程现在等待 s1 锁定的解除。因为它需要等待，所以线程调度器再次调度第一个线程，但第一个线程在等待 s2 锁定的解除。这两个线程现在都在等待，只要锁定块没有结束，就不会解除锁定。这是一个典型的死锁(代码文件 ThreadingIssues/SampleTask.cs)。

```
public class SampleTask
{
  public SampleTask(StateObject s1, StateObject s2)
  {
    _s1 = s1;
    _s2 = s2;
  }
  private StateObject _s1;
  private StateObject _s2;

  public void Deadlock1()
  {
    int i = 0;
    while (true)
    {
      lock (_s1)
      {
        lock (_s2)
        {
          _s1.ChangeState(i);
          _s2.ChangeState(i++);
          Console.WriteLine($"still running, {i}");
        }
      }
    }
  }

  public void Deadlock2()
  {
    int i = 0;
    while (true)
    {
      lock (_s2)
      {
        lock (_s1)
        {
          _s1.ChangeState(i);
          _s2.ChangeState(i++);
          Console.WriteLine($"still running, {i}");
        }
      }
    }
  }
}
```

结果是，程序运行了许多次循环，不久就没有响应了。“仍在运行”的消息仅写入控制台中几次。同样，死锁问题的发生频率也取决于系统配置，每次运行的结果都不同。

死锁问题并不总是像这样那么明显。一个线程锁定了 s1，接着锁定 s2；另一个线程锁定了 s2，接着锁定 s1。在本例中只需要改变锁定顺序，这两个线程就会以相同的顺序进行锁定。但是，在较大的应用程序中，锁定可能隐藏在方法的深处。为了避免这个问题，可以在应用程序的体系架构中，从一开始就设计好锁定顺序，也可以为锁定定义超时时间。如何定义超时时间详见下一节的内容。

## 21.8 lock 语句和线程安全

C#为多个线程的同步提供了自己的关键字：lock 语句。lock 语句是设置锁定和解除锁定的一种简单方式。在添加 lock 语句之前，先进入另一个争用条件。SharedState 类说明了如何使用线程之间的共享状态，并共享一个整数值(代码文件 SynchronizationSamples/SharedState.cs)。

```
public class SharedState
{
  public int State { get; set; }
}
```

下述所有同步示例的代码(SingletonWPF 除外)都使用如下名称空间：

System
System.Collections.Generic
System.Linq
System.Text
System.Threading
System.Threading.Tasks

Job 类包含 DoTheJob()方法，该方法是新任务的入口点。通过其实现代码，将 SharedState 对象的 State 递增 50 000 次。sharedState 变量在这个类的构造函数中初始化(代码文件 SynchronizationSamples/Job.cs)：

```
public class Job
{
  private SharedState _sharedState;
  public Job(SharedState sharedState)
  {
    _sharedState = sharedState;
  }

  public void DoTheJob()
  {
    for (int i = 0; i < 50000; i++)
    {
      _sharedState.State += 1;
    }
  }
}
```

在 Main()方法中，创建一个 SharedState 对象，并把它传递给 20 个 Task 对象的构造函数。在启动所有的任务后，Main()方法进入另一个循环，等待 20 个任务都执行完毕。任务执行完毕后，把共享状态的合计值写入控制台中。因为执行了 50 000 次循环，有 20 个任务，所以写入控制台的值应是 1 000 000。但是，事实常常并非如此(代码文件 SynchronizationSamples/Program.cs)。

```
class Program
{
  static void Main()
  {
    int numTasks = 20;
    var state = new SharedState();
    var tasks = new Task[numTasks];
    for (int i = 0; i < numTasks; i++)
    {
      tasks[i] = Task.Run(() => new Job(state).DoTheJob());
    }
```

```
    Task.WaitAll(tasks);
    Console.WriteLine($"summarized {state.State}");
  }
}
```

多次运行应用程序的结果如下所示：

```
summarized 424687
summarized 465708
summarized 581754
summarized 395571
summarized 633601
```

每次运行的结果都不同，但没有一个结果是正确的。如前所述，调试版本和发布版本的区别很大。根据使用的 CPU 类型，结果也不一样。如果将循环次数改为比较小的值，就会多次得到正确的值，但不是每次都正确。这个应用程序非常小，很容易看出问题，但该问题的原因在大型应用程序中就很难确定。

必须在这个程序中添加同步功能，这可以用 lock 关键字实现。用 lock 语句定义的对象表示，要等待指定对象的锁定。只能传递引用类型。锁定值类型只是锁定了一个副本，这没有什么意义。如果对值类型使用了 lock 语句，C#编译器就会发出一个错误。进行了锁定后——只锁定了一个线程，就可以运行 lock 语句块。在 lock 语句块的最后，对象的锁定被解除，另一个等待锁定的线程就可以获得该锁定块了。

```
lock (obj)
{
  // synchronized region
}
```

要锁定静态成员，可以把锁放在 object 类型或静态成员上：

```
lock (typeof(StaticClass))
{
}
```

使用 lock 关键字可以将类的实例成员设置为线程安全的。这样，一次只有一个线程能访问相同实例的 DoThis()和 DoThat()方法。

```
public class Demo
{
  public void DoThis()
  {
    lock (this)
    {
      // only one thread at a time can access the DoThis and DoThat methods
    }
  }

  public void DoThat()
  {
    lock (this)
    {
    }
  }
}
```

但是，因为实例的对象也可以用于外部的同步访问，而且我们不能在类自身中控制这种访问，所以应采用 SyncRoot 模式。通过 SyncRoot 模式，创建一个私有对象_syncRoot，将这个对象用于 lock 语句。

```
public class Demo
{
  private object _syncRoot = new object();
  public void DoThis()
  {
    lock (_syncRoot)
    {
      // only one thread at a time can access the DoThis and DoThat methods
    }
  }

  public void DoThat()
  {
    lock (_syncRoot)
    {
    }
  }
}
```

使用锁定需要时间，且并不总是必要的。可以创建类的两个版本，一个同步版本，一个异步版本。下一个示例通过修改 Demo 类来说明。Demo 类本身并不是同步的，这可以在 DoThis()和 DoThat()方法的实现中看出。该类还定义了 IsSynchronized 属性，客户可以从该属性中获得类的同步选项信息。为了获得该类的同步版本，可以使用静态方法 Synchronized()传递一个非同步对象，这个方法会返回 SynchronizedDemo 类型的对象。SynchronizedDemo 实现为派生自基类 Demo 的一个内部类，并重写基类的虚成员。重写的成员使用了 SyncRoot 模式。

```
public class Demo
{
  private class SynchronizedDemo: Demo
  {
    private object _syncRoot = new object();
    private Demo _d;
    public SynchronizedDemo(Demo d)
    {
      _d = d;
    }

    public override bool IsSynchronized => true;

    public override void DoThis()
    {
      lock (_syncRoot)
      {
        _d.DoThis();
      }
    }

    public override void DoThat()
    {
      lock (_syncRoot)
      {
        _d.DoThat();
      }
    }
  }

  public virtual bool IsSynchronized => false;

  public static Demo Synchronized(Demo d)
  {
    if (!d.IsSynchronized)
    {
      return new SynchronizedDemo(d);
    }
    return d;
  }

  public virtual void DoThis()
  {
  }

  public virtual void DoThat()
  {
  }
}
```

必须注意，在使用 SynchronizedDemo 类时，只有方法是同步的。对这个类的两个成员的调用并没有同步。

首先修改异步的 SharedState 类，以使用 SyncRoot 模式。如果试图用 SyncRoot 模式锁定对属性的访问，使 SharedState 类变成线程安全的，就仍会出现前面描述的争用条件。

```
public class SharedState
{
  private int _state = 0;
  private object _syncRoot = new object();
  public int State // there's still a race condition,
  // don't do this!
  {
    get { lock (_syncRoot) { return _state; }}
    set { lock (_syncRoot) { _state = value; }}
  }
}
```

调用方法 DoTheJob()的线程访问 SharedState 类的 get 存取器，以获得 state 的当前值，接着 get 存取器给 state 设置新值。在调用对象的 get 和 set 存取器期间，对象没有锁定，另一个线程可以获得临时值(代码文件 SynchronizationSamples/Job.cs)。

```
public void DoTheJob()
{
  for (int i = 0; i < 50000; i++)
  {
    _sharedState.State += 1;
  }
}
```

所以，最好不改变 SharedState 类，让它依旧没有线程安全性(代码文件 SynchronizationSamples/SharedState.cs)。

```
public class SharedState
{
  public int State { get; set; }
}
```

然后在 DoTheJob 方法中，将 lock 语句添加到合适的地方(代码文件 SynchronizationSamples/Job.cs)：

```
public void DoTheJob()
{
  for (int i = 0; i < 50000; i++)
  {
    lock (_sharedState)
    {
      _sharedState.State += 1;
    }
  }
}
```

这样，应用程序的结果就总是正确的。

**注意：**
*在一个地方使用 lock 语句并不意味着，访问对象的其他线程都正在等待。必须对每个访问共享状态的线程显式地使用同步功能。*

当然，还必须修改 SharedState 类的设计，并作为一个原子操作提供递增方式。这是一个设计问题——把什么实现为类的原子功能？下面的代码片段锁定了递增操作。

```
public class SharedState
{
  private int _state = 0;
  private object _syncRoot = new object();
  public int State => _state;
  public int IncrementState()
  {
    lock (_syncRoot)
    {
      return ++_state;
    }
  }
}
```

锁定状态的递增还有一种更快的方式，如下节所示。

## 21.9 Interlocked 类

Interlocked 类用于使变量的简单语句原子化。i++不是线程安全的，它的操作包括从内存中获取一个值，给该值递增 1，再将它存储回内存。这些操作都可能会被线程调度器打断。Interlocked 类提供了以线程安全的方式递增、递减、交换和读取值的方法。

与其他同步技术相比，使用 Interlocked 类会快得多。但是，它只能用于简单的同步问题。

例如，这里不使用 lock 语句锁定对 someState 变量的访问，把它设置为一个新值，以防它是空的，而可以

使用 Interlocked 类，它比较快：

```
lock (this)
{
  if (_someState == null)
  {
    _someState = newState;
  }
}
```

这个功能相同但比较快的版本使用了 Interlocked.CompareExchange()方法。

不是像下面这样在 lock 语句中执行递增操作：

```
public int State
{
  get
  {
    lock (this)
    {
      return ++_state;
    }
  }
}
```

而使用较快的 Interlocked.Increment()方法：

```
public int State
{
  get => Interlocked.Increment(ref _state);
}
```

# 21.10 Monitor 类

lock 语句由 C#编译器解析为使用 Monitor 类。下面的 lock 语句：

```
lock (obj)
{
  // synchronized region for obj
}
```

被解析为调用 Enter()方法，该方法会一直等待，直到线程锁定对象为止。一次只有一个线程能锁定对象。只要解除了锁定，线程就可以进入同步阶段。Monitor 类的 Exit()方法解除了锁定。编译器把 Exit()方法放在 try 块的 finally 处理程序中，所以如果抛出了异常，就会解除该锁定。

**注意：**
try/finally 块详见第 14 章。

```
Monitor.Enter(obj);
try
{
  // synchronized region for obj
}
finally
{
  Monitor.Exit(obj);
}
```

与 C#的 lock 语句相比，Monitor 类的主要优点是：可以添加一个等待被锁定的超时值。这样就不会无限期地等待被锁定，而可以像下面的例子那样使用 TryEnter()方法，其中给它传递一个超时值，指定等待被锁定的最长时间。如果 obj 被锁定，TryEnter()方法就把布尔型的引用参数设置为 true，并同步地访问由对象 obj 锁定的状态。如果另一个线程锁定 obj 的时间超过了 500 毫秒，TryEnter()方法就把变量 lockTaken 设置为 false，线程不再等待，而是用于执行其他操作。也许在以后，该线程会尝试再次获得锁定。

```
bool _lockTaken = false;
Monitor.TryEnter(_obj, 500, ref _lockTaken);
if (_lockTaken)
  {
  try
```

```
    {
      // acquired the lock
      // synchronized region for obj
    }
    finally
    {
      Monitor.Exit(obj);
    }
  }
  else
  {
    // didn't get the lock, do something else
  }
```

## 21.11　SpinLock 结构

如果基于对象的锁定对象(Monitor)的系统开销由于垃圾收集而过高，就可以使用 SpinLock 结构。如果有大量的锁定(例如，列表中的每个节点都有一个锁定)，且锁定的时间总是非常短，SpinLock 结构就很有用。应避免使用多个 SpinLock 结构，也不要调用任何可能阻塞的内容。

除了体系结构上的区别之外，SpinLock 结构的用法非常类似于 Monitor 类。使用 Enter()或 TryEnter()方法获得锁定，使用 Exit()方法释放锁定。SpinLock 结构还提供了属性 IsHeld 和 IsHeldByCurrentThread，指定它当前是否是锁定的。

> **注意：**
> 传送 SpinLock 实例时要小心。因为 SpinLock 定义为结构，把一个变量赋予另一个变量会创建一个副本。总是通过引用传送 SpinLock 实例。

## 21.12　WaitHandle 基类

WaitHandle 是一个抽象基类，用于等待一个信号的设置。可以等待不同的信号，因为 WaitHandle 是一个基类，可以从中派生一些类。

使用 WaitHandle 基类可以等待一个信号的出现(WaitOne()方法)、等待必须发出信号的多个对象(WaitAll()方法)，或者等待多个对象中的一个(WaitAny()方法)。WaitAll()和 WaitAny()是 WaitHandle 类的静态方法，接收一个 WaitHandle 参数数组。

WaitHandle 基类有一个 SafeWaitHandle 属性，其中可以将一个本机句柄赋予一个操作系统资源，并等待该句柄。例如，可以指定一个 SafeFileHandle 等待文件 I/O 操作的完成。

因为 Mutex、EventWaitHandle 和 Semaphore 类派生自 WaitHandle 基类，所以可以在等待时使用它们。

## 21.13　Mutex 类

Mutex(mutual exclusion，互斥)是.NET Framework 中提供跨多个进程同步访问的一个类。它非常类似于 Monitor 类，因为它们都只有一个线程能拥有锁定。只有一个线程能获得互斥锁定，访问受互斥保护的同步代码区域。

在 Mutex 类的构造函数中，可以指定互斥是否最初应由主调线程拥有，定义互斥的名称，获得互斥是否已存在的信息。在下面的示例代码中，第 3 个参数定义为输出参数，接收一个表示互斥是否为新建的布尔值。如果返回的值是 false，就表示互斥已经定义。互斥可以在另一个进程中定义，因为操作系统能够识别有名称的互斥，它由不同的进程共享。如果没有给互斥指定名称，互斥就是未命名的，不在不同的进程之间共享。

```
bool createdNew;
var mutex = new Mutex(false, "ProCSharpMutex", out createdNew);
```

要打开已有的互斥，还可以使用 Mutex.OpenExisting()方法，它不需要用构造函数创建互斥时需要的相

同.NET 权限。

由于 Mutex 类派生自基类 WaitHandle，因此可以利用 WaitOne()方法获得互斥锁定，在该过程中成为该互斥的拥有者。通过调用 ReleaseMutex()方法，即可释放互斥。

```
if (mutex.WaitOne())
{
  try
  {
  // synchronized region
  }
  finally
  {
    mutex.ReleaseMutex();
  }
}
else
{
  // some problem happened while waiting
}
```

由于系统能识别有名称的互斥，因此可以使用它禁止应用程序启动两次。在下面的控制台应用程序中，调用了 Mutex 对象的构造函数。接着，验证名称为 SingletonAppMutex 的互斥是否存在。如果存在，应用程序就退出(代码文件 SingletonUsingMutex/Program.cs)。

```
static void Main()
{
  bool mutexCreated;
  var mutex = new Mutex(false, "SingletonAppMutex", out mutexCreated);
  if (!mutexCreated)
  {
    Console.WriteLine("You can only start one instance of the application.");
    Console.WriteLine("Exiting.");
    return;
  }
  Console.WriteLine("Application running");
  Console.WriteLine("Press return to exit");
  Console.ReadLine();
}
```

# 21.14 Semaphore 类

信号量非常类似于互斥，其区别是，信号量可以同时由多个线程使用。信号量是一种计数的互斥锁定。使用信号量，可以定义允许同时访问受旗语锁定保护的资源的线程个数。如果需要限制可以访问可用资源的线程数，信号量就很有用。例如，如果系统有 3 个物理端口可用，就允许 3 个线程同时访问 I/O 端口，但第 4 个线程需要等待前 3 个线程中的一个释放资源。

.NET Core 为信号量功能提供了两个类 Semaphore 和 SemaphoreSlim。Semaphore 类可以命名，使用系统范围内的资源，允许在不同进程之间同步。SemaphoreSlim 类是对较短等待时间进行了优化的轻型版本。

在下面的示例应用程序中，在 Main()方法中创建了 6 个任务和一个计数为 3 的信号量。在 Semaphore 类的构造函数中，定义了锁定个数的计数，它可以用信号量(第二个参数)来获得，还定义了最初释放的锁定数(第一个参数)。如果第一个参数的值小于第二个参数，它们的差就是已经分配线程的计数值。与互斥一样，也可以给信号量指定名称，使之在不同的进程之间共享。这里定义信号量时没有指定名称，所以它只能在这个进程中使用。在创建了 SemaphoreSlim 对象之后，启动 6 个任务，它们都获得了相同的信号量(代码文件 SemaphoreSample/Program.cs)。

```
class Program
{
  static void Main()
  {
    int taskCount = 6;
    int semaphoreCount = 3;
    var semaphore = new SemaphoreSlim(semaphoreCount, semaphoreCount);
    var tasks = new Task[taskCount];
    for (int i = 0; i < taskCount; i++)
```

```
      {
        tasks[i] = Task.Run(() => TaskMain(semaphore));
      }
      Task.WaitAll(tasks);
      Console.WriteLine("All tasks finished");
    }
    //...
```

在任务的主方法 TaskMain()中，任务利用 Wait()方法锁定信号量。信号量的计数是 3，所以有 3 个任务可以获得锁定。第 4 个任务必须等待，这里还定义了最长的等待时间为 600 毫秒。如果在该等待时间过后未能获得锁定，任务就把一条消息写入控制台，在循环中继续等待。只要获得了锁定，线程就把一条消息写入控制台，睡眠一段时间，然后解除锁定。在解除锁定时，在任何情况下一定要解除资源的锁定，这一点很重要。这就是在 finally 处理程序中调用 SemaphoreSlim 类的 Release()方法的原因(代码文件 SemaphoreSample/Program.cs)。

```
// ...
public static void TaskMain(SemaphoreSlim semaphore)
{
  bool isCompleted = false;
  while (!isCompleted)
  {
    if (semaphore.Wait(600))
    {
      try
      {
        Console.WriteLine($"Task {Task.CurrentId} locks the semaphore");
        Task.Delay(2000).Wait();
      }
      finally
      {
        Console.WriteLine($"Task {Task.CurrentId} releases the semaphore");
        semaphore.Release();
        isCompleted = true;
      }
    }
    else
    {
      Console.WriteLine($"Timeout for task {Task.CurrentId}; wait again");
    }
  }
}
```

运行应用程序，可以看到有 4 个线程很快被锁定。ID 为 7、8 和 9 的线程需要等待。该等待会重复进行，直到其中一个被锁定的线程解除了信号量。

```
Task 4 locks the semaphore
Task 5 locks the semaphore
Task 6 locks the semaphore
Timeout for task 7; wait again
Timeout for task 7; wait again
Timeout for task 8; wait again
Timeout for task 7; wait again
Timeout for task 8; wait again
Timeout for task 7; wait again
Timeout for task 9; wait again
Timeout for task 8; wait again
Task 5 releases the semaphore
Task 7 locks the semaphore
Task 6 releases the semaphore
Task 4 releases the semaphore
Task 8 locks the semaphore
Task 9 locks the semaphore
Task 8 releases the semaphore
Task 7 releases the semaphore
Task 9 releases the semaphore
All tasks finished
```

## 21.15　Events 类

与互斥和信号量对象一样，事件也是一个系统范围内的资源同步方法。为了从托管代码中使用系统事件，.NET Framework 在 System.Threading 名称空间中提供了 ManualResetEvent、AutoResetEvent、ManualResetEventSlim 和

CountdownEvent 类。

**注意：**
第 8 章介绍了 C#中的 event 关键字，它与 System.Threading 名称空间中的 event 类没有关系。event 关键字基于委托，而上述两个 event 类是.NET 封装器，用于系统范围内的本机事件资源的同步。

可以使用事件通知其他任务：这里有一些数据，并完成了一些操作等。事件可以发信号，也可以不发信号。使用前面介绍的 WaitHandle 类，任务可以等待处于发信号状态的事件。

调用 Set()方法，即可向 ManualResetEventSlim 发信号。调用 Reset()方法，可以使之返回不发信号的状态。如果多个线程等待向一个事件发信号，并调用了 Set()方法，就释放所有等待的线程。另外，如果一个线程刚刚调用了 WaitOne()方法，但事件已经发出信号，等待的线程就可以继续等待。

也通过调用 Set()方法向 AutoResetEvent 发信号。也可以使用 Reset()方法使之返回不发信号的状态。但是，如果一个线程在等待自动重置的事件发信号，当第一个线程的等待状态结束时，该事件会自动变为不发信号的状态。这样，如果多个线程在等待向事件发信号，就只有一个线程结束其等待状态，它不是等待时间最长的线程，而是优先级最高的线程。

为了说明 ManualResetEventSlim 类的事件，下面的 Calculator 类定义了 Calculation()方法，这是任务的入口点。在这个方法中，该任务接收用于计算的输入数据，将结果写入变量 result，该变量可以通过 Result 属性来访问。只要完成了计算(在随机的一段时间过后)，就调用 ManualResetEventSlim 类的 Set 方法，向事件发信号(代码文件 EventSample/Calculator.cs)。

```
public class Calculator
{
  private ManualResetEventSlim _mEvent;
  public int Result { get; private set; }

  public Calculator(ManualResetEventSlim ev)
  {
    _mEvent = ev;
  }

  public void Calculation(int x, int y)
  {
    Console.WriteLine($"Task {Task.CurrentId} starts calculation");
    Task.Delay(new Random().Next(3000)).Wait();
    Result = x + y;
    // signal the event-completed!
    Console.WriteLine($"Task {Task.CurrentId} is ready");
    _mEvent.Set();
  }
}
```

程序的 Main()方法定义了包含 4 个 ManualResetEventSlim 对象的数组和包含 4 个 Calculator 对象的数组。每个 Calculator 在构造函数中用一个 ManualResetEventSlim 对象初始化，这样每个任务在完成时都有自己的事件对象来发信号。现在使用 Task 类，让不同的任务执行计算任务(代码文件 EventSample/Program.cs)。

```
class Program
{
  static void Main()
  {
    const int taskCount = 4;
    var mEvents = new ManualResetEventSlim[taskCount];
    var waitHandles = new WaitHandle[taskCount];
    var calcs = new Calculator[taskCount];
    for (int i = 0; i < taskCount; i++)
    {
      int i1 = i;
      mEvents[i] = new ManualResetEventSlim(false);
      waitHandles[i] = mEvents[i].WaitHandle;
      calcs[i] = new Calculator(mEvents[i]);
      Task.Run(() => calcs[i1].Calculation(i1 + 1, i1 + 3));
    }
  //...
```

WaitHandle 类现在用于等待数组中的任意一个事件。WaitAny()方法等待向任意一个事件发信号。与

ManualResetEvent 对象不同，ManualResetEventSlim 对象不派生自 WaitHandle 类。因此有一个 WaitHandle 对象的集合，它在 ManualResetEventSlim 类的 WaitHandle 属性中填充。从 WaitAny()方法返回的 index 值匹配传递给 WaitAny()方法的事件数组的索引，以提供发信号的事件的相关信息，使用该索引可以从这个事件中读取结果。

```
for (int i = 0; i < taskCount; i++)
{
  int index = WaitHandle.WaitAny(waitHandles);
  if (index == WaitHandle.WaitTimeout)
  {
    Console.WriteLine("Timeout!!");
  }
  else
  {
    mEvents[index].Reset();
    Console.WriteLine($"finished task for {index}, result:
      {calcs[index].Result}");
  }
}
```

启动应用程序时，可以看到任务在进行计算并设置事件，以通知主线程，它可以读取结果了。在任意时间，依据是调试版本还是发布版本，以及硬件的不同，会看到执行调用的任务有不同的顺序和不同的数量。

```
Task 4 starts calculation
Task 5 starts calculation
Task 6 starts calculation
Task 7 starts calculation
Task 7 is ready
finished task for 3, result: 10
Task 4 is ready
finished task for 0, result: 4
Task 6 is ready
finished task for 1, result: 6
Task 5 is ready
finished task for 2, result: 8
```

在一个类似的场景中，为了把一些工作分支到多个任务中，并在以后合并结果，使用新的 CountdownEvent 类很有用。不需要为每个任务创建一个单独的事件对象，而只需要创建一个事件对象。CountdownEvent 类为所有设置了事件的任务定义一个初始数字，在到达该计数后，就向 CountdownEvent 类发信号。

修改 Calculator 类，以使用 CountdownEvent 类替代 ManualResetEvent 类。不使用 Set()方法设置信号，而使用 CountdownEvent 类定义 Signal()方法(代码文件 EventSampleWithCountdownEvent /Calculator.cs)。

```
public class Calculator
{
  private CountdownEvent _cEvent;
  public int Result { get; private set; }

  public Calculator(CountdownEvent ev)
  {
    _cEvent = ev;
  }

  public void Calculation(int x, int y)
  {
    Console.WriteLine($"Task {Task.CurrentId} starts calculation");
    Task.Delay(new Random().Next(3000)).Wait();
    Result = x + y;
    // signal the event-completed!
    Console.WriteLine($"Task {Task.CurrentId} is ready");
    _cEvent.Signal();
  }
}
```

Main()方法现在可以简化，使它只需要等待一个事件。如果不像前面那样单独处理结果，这个新版本就很不错。

```
const int taskCount = 4;
var cEvent = new CountdownEvent(taskCount);
var calcs = new Calculator[taskCount];
for (int i = 0; i < taskCount; i++)
{
  calcs[i] = new Calculator(cEvent);
  int i1 = i;
```

```
  Task.Run(() => calcs[i1].Calculation, Tuple.Create(i1 + 1, i1 + 3));
}
cEvent.Wait();

Console.WriteLine("all finished");
for (int i = 0; i < taskCount; i++)
{
  Console.WriteLine($"task for {i}, result: {calcs[i].Result}");
}
```

# 21.16 Barrier 类

对于同步，Barrier 类非常适用于其中工作有多个任务分支且以后又需要合并工作的情况。Barrier 类用于需要同步的参与者。激活一个任务时，就可以动态地添加其他参与者，例如，从父任务中创建子任务。参与者在继续之前，可以等待所有其他参与者完成其工作。

BarrierSample 有点复杂，但它展示了 Barrier 类型的功能。下面的应用程序使用一个包含 2 000 000 个随机字符串的集合。使用多个任务遍历该集合，并统计以 a、b、c 等开头的字符串个数。工作不仅分布在不同的任务之间，也放在一个任务中。毕竟所有的任务都迭代字符串的第一个集合，汇总结果，以后任务会继续处理下一个集合。

FillData()方法创建一个集合，并用随机字符串填充它(代码文件 BarrierSample/Program.cs)：

```
public static IEnumerable<string> FillData(int size)
{
  var r = new Random();
  return Enumerable.Range(0, size).Select(x => GetString(r));
}

private static string GetString(Random r)
{
  var sb = new StringBuilder(6);
  for (int i = 0; i < 6; i++)
  {
    sb.Append((char)(r.Next(26) + 97));
  }
  return sb.ToString();
}
```

在 LogBarrierInformation 方法中定义一个辅助方法，来显示 Barrier 的信息：

```
private static void LogBarrierInformation(string info, Barrier barrier)
{
  Console.WriteLine($"Task {Task.CurrentId}: {info}. " +
    $"{barrier.ParticipantCount} current and " +
    $"{barrier.ParticipantsRemaining} remaining participants, " +
    $"phase {barrier.CurrentPhaseNumber}");
}
```

CalculationInTask()方法定义了任务执行的作业。通过参数，第 3 个参数引用 Barrier 实例。用于计算的数据是数组 IList<string>。最后一个参数是 int 锯齿数组，用于在任务执行过程中写出结果。

任务把处理放在一个循环中。每一次循环中，都处理 IList<string>[]的数组元素。每个循环完成后,任务通过调用 SignalAndWait 方法，发出做好了准备的信号，并等待，直到所有的其他任务也准备好处理为止。这个循环会继续执行，直到任务完全完成为止。接着，任务就会使用 RemoveParticipant()方法从 Barrier 类中删除它自己(代码文件 BarrierSample/Program.cs)：

```
private static void CalculationInTask(int jobNumber, int partitionSize,
  Barrier barrier, IList<string>[] coll, int loops, int[][] results)
{
  LogBarrierInformation("CalculationInTask started", barrier);
  for (int i = 0; i < loops; i++)
  {
    var data = new List<string>(coll[i]);
    int start = jobNumber * partitionSize;
    int end = start + partitionSize;
    Console.WriteLine($"Task {Task.CurrentId} in loop {i}: partition " +
      $"from {start} to {end}");
    for (int j = start; j < end; j++)
```

```
      {
        char c = data[j][0];
        results[i][c - 97]++;
      }
      Console.WriteLine($"Calculation completed from task {Task.CurrentId} " +
        $"in loop {i}. {results[i][0]} times a, {results[i][25]} times z");
      LogBarrierInformation("sending signal and wait for all", barrier);
      barrier.SignalAndWait();
      LogBarrierInformation("waiting completed", barrier);
    }
    barrier.RemoveParticipant();
    LogBarrierInformation("finished task, removed participant", barrier);
  }
```

在 Main()方法中创建一个 Barrier 实例。在构造函数中，可以指定参与者的数量。在该示例中，这个数量是 3(numberTasks + 1)，因为该示例创建了两个任务，Main()方法本身也是一个参与者。使用 Task.Run 创建两个任务，把遍历集合的任务分为两个部分。启动该任务后，使用 SignalAndWait()方法，Main()方法在完成时发出信号，并等待所有其他参与者或者发出完成的信号，或者从 Barrier 类中删除它们。一旦所有的参与者都准备好，就提取任务的结果，并使用 Zip()扩展方法把它们合并起来。接着进行下一次迭代，等待任务的下一个结果(代码文件 BarrierSample/Program.cs)：

```
static void Main()
{
  const int numberTasks = 2;
  const int partitionSize = 1000000;
  const int loops = 5;
  var taskResults = new Dictionary<int, int[][]>();
  var data = new List<string>[loops];
  for (int i = 0; i < loops; i++)
  {
    data[i] = new List<string>(FillData(partitionSize * numberTasks);
  }

  var barrier = new Barrier(numberTasks + 1);
  LogBarrierInformation("initial participants in barrier", barrier);
  for (int i = 0; i < numberTasks; i++)
  {
    barrier.AddParticipant();
    int jobNumber = i;
    taskResults.Add(i, new int[loops][]);
    for (int loop = 0; loop < loops; loop++)
    {
      taskResult[i, loop] = new int[26];
    }
    Console.WriteLine("Main - starting task job {jobNumber}");
    Task.Run(() => CalculationInTask(jobNumber, partitionSize,
      barrier, data, loops, taskResults[jobNumber]));
  }

  for (int loop = 0; loop < 5; loop++)
  {
    LogBarrierInformation("main task, start signaling and wait", barrier);
    barrier.SignalAndWait();
    LogBarrierInformation("main task waiting completed", barrier);
    int[][] resultCollection1 = taskResults[0];
    int[][] resultCollection2 = taskResults[1];
    var resultCollection = resultCollection1[loop].Zip(
      resultCollection2[loop], (c1, c2) => c1 + c2);
    char ch = 'a';
    int sum = 0;
    foreach (var x in resultCollection)
    {
      Console.WriteLine($"{ch++}, count: {x}");
      sum += x;
    }
    LogBarrierInformation($"main task finished loop {loop}, sum: {sum}",
      barrier);
  }

  Console.WriteLine("finished all iterations");
  Console.ReadLine();
}
```

**注意：**
锯齿数组参见第 7 章，zip 扩展方法参见第 13 章。

运行应用程序，输出如下所示。在输出中可以看到，每个 AddParticipant 调用都会增加参与者的数量和剩下的参与者数量。只要一个参与者调用 SignalAndWait，剩下的参与者数就会递减。当剩下的参与者数量达到 0 时，所有参与者的等待就结束，开始下一个阶段：

```
Task : initial participants in barrier. 1 current and 1 remaining participants,
phase 0.
Main - starting task job 0
Main - starting task job 1
Task : main task, starting signaling and wait. 3 current and
3 remaining participants, phase 0.
Task 4: CalculationInTask started. 3 current and 2 remaining participants, phase 0.
Task 5: CalculationInTask started. 3 current and 2 remaining participants, phase 0.
Task 4 in loop 0: partition from 0 to 1000000
Task 5 in loop 0: partition from 1000000 to 2000000
Calculation completed from task 4 in loop 0. 38272 times a, 38637 times z
Task 4: sending signal and wait for all. 3 current and
2 remaining participants, phase 0.
Calculation completed from task 5 in loop 0. 38486 times a, 38781 times z
Task 5: sending signal and wait for all. 3 current and
1 remaining participants, phase 0.
Task 5: waiting completed. 3 current and 3 remaining participants, phase 1
Task 4: waiting completed. 3 current and 3 remaining participants, phase 1
Task : main waiting completed. 3 current and 3 remaining participants, phase 1
```

## 21.17 ReaderWriterLockSlim 类

为了使锁定机制允许锁定多个读取器(而不是一个写入器)访问某个资源，可以使用 ReaderWriterLockSlim 类。这个类提供了一个锁定功能，如果没有写入器锁定资源，就允许多个读取器访问资源，但只能有一个写入器锁定该资源。

ReaderWriterLockSlim 类有阻塞或不阻塞的方法来获取读取锁，如阻塞的 EnterReadLock()和不阻塞的 TryEnterReadLock()方法，还可以使用阻塞的 EnterWriteLock()和不阻塞的 TryEnterWriteLock()方法获得写入锁定。如果任务先读取资源，之后写入资源，它就可以使用 EnterUpgradableReadLock()或 TryEnterUpgradableReadLock()方法获得可升级的读取锁定。有了这个锁定，就可以获得写入锁定，而不需要释放读取锁定。

这个类的几个属性提供了当前锁定的相关信息，如 CurrentReadCount、WaitingReadCount、WaitingUpgradableReadCount 和 WaitingWriteCount。

下面的示例程序创建了一个包含 6 项的集合和一个 ReaderWriterLockSlim()对象。ReaderMethod()方法获得一个读取锁定，读取列表中的所有项，并把它们写到控制台中。WriterMethod()方法试图获得一个写入锁定，以改变集合的所有值。在 Main()方法中，启动 6 个任务，以调用 ReaderMethod()或 WriterMethod()方法(代码文件 ReaderWriterSample/Program.cs)。

```
class Program
{
  private static List<int> _items = new List<int>() { 0, 1, 2, 3, 4, 5};
  private static ReaderWriterLockSlim _rwl =
    new ReaderWriterLockSlim(LockRecursionPolicy.SupportsRecursion);

  public static void ReaderMethod(object reader)
  {
    try
    {
      _rwl.EnterReadLock();
      for (int i = 0; i < _items.Count; i++)
      {
        Console.WriteLine($"reader {reader}, loop: {i}, item: {_items[i]}");
        Task.Delay(40).Wait();
      }
    }
    finally
    {
```

```
            _rwl.ExitReadLock();
          }
        }

        public static void WriterMethod(object writer)
        {
          try
          {
            while (!_rwl.TryEnterWriteLock(50))
            {
              Console.WriteLine($"Writer {writer} waiting for the write lock");
              Console.WriteLine($"current reader count: {_rwl.CurrentReadCount}");
            }
            Console.WriteLine($"Writer {writer} acquired the lock");
            for (int i = 0; i < _items.Count; i++)
            {
              _items[i]++;
              Task.Delay(50).Wait();
            }
            Console.WriteLine($"Writer {writer} finished");
          }
          finally
          {
            _rwl.ExitWriteLock();
          }
        }

        static void Main()
        {
          var taskFactory = new TaskFactory(TaskCreationOptions.LongRunning,
            TaskContinuationOptions.None);
          var tasks = new Task[6];
          tasks[0] = taskFactory.StartNew(WriterMethod, 1);
          tasks[1] = taskFactory.StartNew(ReaderMethod, 1);
          tasks[2] = taskFactory.StartNew(ReaderMethod, 2);
          tasks[3] = taskFactory.StartNew(WriterMethod, 2);
          tasks[4] = taskFactory.StartNew(ReaderMethod, 3);
          tasks[5] = taskFactory.StartNew(ReaderMethod, 4);
          Task.WaitAll(tasks);
        }
      }
```

运行这个应用程序，可以看到第一个写入器先获得锁定。第二个写入器和所有的读取器需要等待。接着，读取器可以同时工作，而第二个写入器仍在等待资源。

```
Writer 1 acquired the lock
Writer 2 waiting for the write lock
current reader count: 0
Writer 2 waiting for the write lock
current reader count: 0
Writer 2 waiting for the write lock
current reader count: 0
Writer 2 waiting for the write lock
current reader count: 0
Writer 1 finished
reader 4, loop: 0, item: 1
reader 1, loop: 0, item: 1
Writer 2 waiting for the write lock
current reader count: 4
reader 2, loop: 0, item: 1
reader 3, loop: 0, item: 1
reader 4, loop: 1, item: 2
reader 1, loop: 1, item: 2
reader 3, loop: 1, item: 2
reader 2, loop: 1, item: 2
Writer 2 waiting for the write lock
current reader count: 4
reader 4, loop: 2, item: 3
reader 1, loop: 2, item: 3
reader 2, loop: 2, item: 3
reader 3, loop: 2, item: 3
Writer 2 waiting for the write lock
current reader count: 4
reader 4, loop: 3, item: 4
reader 1, loop: 3, item: 4
reader 2, loop: 3, item: 4
reader 3, loop: 3, item: 4
```

```
reader 4, loop: 4, item: 5
reader 1, loop: 4, item: 5
Writer 2 waiting for the write lock
current reader count: 4
reader 2, loop: 4, item: 5
reader 3, loop: 4, item: 5
reader 4, loop: 5, item: 6
reader 1, loop: 5, item: 6
reader 2, loop: 5, item: 6
reader 3, loop: 5, item: 6
Writer 2 waiting for the write lock
current reader count: 4
Writer 2 acquired the lock
Writer 2 finished
```

## 21.18 Lock 和 await

如果试图在 lock 块中使用 async 关键字时使用 lock 关键字，会得到这个编译错误：cannot await in the body of a lock statement。原因是在 async 完成之后，该方法可能会在一个不同的线程中运行，而不是在 async 关键字之前。lock 关键字需要同一个线程中获取锁和释放锁。

下面的代码块会导致编译错误：

```
static async Task IncorrectLockAsync()
{
  lock (s_syncLock)
  {
    Console.WriteLine($"{nameof(IncorrectLockAsync)} started");
    await Task.Delay(500);  // compiler error: cannot await in the body
      // of a lock statement
    Console.WriteLine($"{nameof(IncorrectLockAsync)} ending");
  }
}
```

如何解决这个问题？不能为此使用 Monitor，因为 Monitor 需要从它获取锁的同一线程中释放锁。lock 关键字基于 Monitor。

虽然 Mutex 对象可以用于不同进程之间的同步，但它有相同的问题：它为线程授予了一个锁。从不同的线程中释放锁是不可能的。相反，可以使用 Semaphore 或 SemaphoreSlim 类。Semaphore 可以从不同的线程中释放信号量。

下面的代码片段使用 SemaphoreSlim 对象上的 WaitAsync 等待获得一个信号量。SemaphoreSlim 对象初始化为计数 1，因此对信号量的等待只授予一次。在 finally 代码块中，通过调用 Release 方法释放信号量(代码文件 LockAcrossAwait/Program.cs)：

```
private static SemaphoreSlim s_asyncLock = new SemaphoreSlim(1);
static async Task LockWithSemaphore(string title)
{
  Console.WriteLine($"{title} waiting for lock");
  await s_asyncLock.WaitAsync();
  try
  {
    Console.WriteLine($"{title} {nameof(LockWithSemaphore)} started");
    await Task.Delay(500);
    Console.WriteLine($"{title} {nameof(LockWithSemaphore)} ending");
  }
  finally
  {
    s_asyncLock.Release();
  }
}
```

下面尝试在多个任务中同时调用此方法。该方法 RunUseSemaphoreAsync 启动 6 个任务，并发地调用 LockWithSemaphore 方法：

```
static async Task RunUseSemaphoreAsync()
{
  Console.WriteLine(nameof(RunUseSemaphoreAsync));
  string[] messages = { "one", "two", "three", "four", "five", "six" };
  Task[] tasks = new Task[messages.Length];
```

```
    for (int i = 0; i < messages.Length; i++)
    {
      string message = messages[i];

      tasks[i] = Task.Run(async () =>
      {
        await LockWithSemaphore(message);
      });
    }

    await Task.WhenAll(tasks);
    Console.WriteLine();
  }
```

运行该程序，可以看到多个任务同时启动，但是在信号量被锁定后，所有其他任务都需要等待信号量再次释放：

```
RunLockWithAwaitAsync
two waiting for lock
two LockWithSemaphore started
three waiting for lock
five waiting for lock
four waiting for lock
six waiting for lock
one waiting for lock
two LockWithSemaphore ending
three LockWithSemaphore started
three LockWithSemaphore ending
five LockWithSemaphore started
five LockWithSemaphore ending
four LockWithSemaphore started
four LockWithSemaphore ending
six LockWithSemaphore started
six LockWithSemaphore ending
one LockWithSemaphore started
one LockWithSemaphore ending
```

为了更容易地使用锁，可以创建一个实现 IDisposable 接口的类来管理资源。对于这个类，可以使用 using 语句，就像使用 lock 状态来锁定和释放信号量一样。

下面的代码片段实现了 AsyncSemaphore 类，该类在构造函数中分配一个 SemaphoreSlim，在 AsyncSemaphore 上调用 WaitAsync 方法时，返回实现接口 IDisposable 的内部类 SemaphoreReleaser。调用 Dispose 方法时，释放信号量(代码文件 LockAcrossAwait /AsyncSemaphore.cs)：

```
public sealed class AsyncSemaphore
{
  private class SemaphoreReleaser : IDisposable
  {
    private SemaphoreSlim _semaphore;

    public SemaphoreReleaser(SemaphoreSlim semaphore) =>
      _semaphore = semaphore;

    public void Dispose() => _semaphore.Release();
  }

  private SemaphoreSlim _semaphore;
  public AsyncSemaphore() =>
    _semaphore = new SemaphoreSlim(1);

  public async Task<IDisposable> WaitAsync()
  {
    await _semaphore.WaitAsync();
    return new SemaphoreReleaser(_semaphore) as IDisposable;
  }
}
```

从前面所示的 LockWithSemaphore 方法中更改实现，现在可以使用 using 语句锁定信号量。记住，using 语句创建一个 catch/finally 块，在 finally 块中调用 Dispose 方法(代码文件 LockAcrossAwait/Program.cs)：

```
private static AsyncSemaphore s_asyncSemaphore = new AsyncSemaphore();
static async Task UseAsyncSemaphore(string title)
{
  using (await s_asyncSemaphore.WaitAsync())
```

```
    {
      Console.WriteLine($"{title} {nameof(LockWithSemaphore)} started");
      await Task.Delay(500);
      Console.WriteLine($"{title} {nameof(LockWithSemaphore)} ending");
    }
  }
```

使用类似于 LockWithSemaphore 方法的 UseAsyncSemaphore 方法会执行相同的行为。然而，类只编写一次，等待过程中的锁定就变得更简单。

## 21.19 小结

本章介绍了如何通过 System.Threading.Tasks 名称空间编写多任务应用程序。在应用程序中使用多线程要仔细规划。太多的线程会导致资源问题，线程不足又会使应用程序执行缓慢，执行效果也不好。使用任务可以获得线程的抽象。这个抽象有助于避免创建过多的线程，因为线程是在池中重用的。

我们探讨了创建多个任务的各种方法，如 Parallel 类。通过使用 Parallel.Invoke、Parallel.ForEach 和 Parallel.For，可以实现任务和数据的并行性。还介绍了如何使用 Task 类来获得对并行编程的全面控制。任务可以在主调线程中异步运行，使用线程池中的线程，以及创建独立的新线程。任务还提供了一个层次结构模型，允许创建子任务，并且提供了一种取消完整层次结构的方法。

取消架构提供了一种标准机制，不同的类可以以相同的方法使用它来提前取消某个任务。

本章讨论了几个可用于.NET 的同步对象，以及适合使用同步对象的场合。简单的同步可以通过 lock 关键字完成。在后台，Monitor 类型允许设置超时，而 lock 关键字不允许。为了在进程之间进行同步，Mutex 对象提供了类似的功能。Semaphore 对象表示带有计数的同步对象，该计数是允许并发运行的任务数量。为了通知其他任务已准备好，讨论了不同类型的事件对象，比如 AutoResetEvent、ManualResetEvent 和 CountdownEvent。拥有多个读取器和写入器的简单方法由 ReaderWriterLock 提供。Barrier 类型提供了一个更复杂的场景，其中可以同时运行多个任务，直到达到一个同步点为止。一旦所有任务达到这一点，它们就可以继续同时满足于下一个同步点。

下面是有关线程的几条规则：

- 尽力使同步要求最低。同步很复杂，且会阻塞线程。如果尝试避免共享状态，就可以避免同步。当然，这不总是可行。
- 类的静态成员应是线程安全的。通常，.NET Framework 中的类满足这个要求。
- 实例状态不需要是线程安全的。为了得到最佳性能，最好在类的外部使用同步功能，且不对类的每个成员使用同步功能。.NET Framework 类的实例成员一般不是线程安全的。在 MSDN API 库中，对于.NET Framework 的每个类在“线程安全性”部分中可以找到相应的归档信息。

第 23 章介绍另一个.NET 核心主题：文件和流。

# 第22章 文件和流

**本章要点**

- 介绍目录结构
- 移动、复制、删除文件和文件夹
- 读写文本文件
- 使用流读写文件
- 使用阅读器和写入器读写文件
- 压缩文件
- 监控文件的变化
- 使用管道进行通信
- 使用 Windows Runtime 流

**本章源代码下载地址(wrox.com)：**

打开 www.wrox.com 的 Download Code 选项卡可下载本章源代码。源代码也可以在 FilesAndStreams 目录的 https://github.com/ProfessionalCSharp/ProfessionalCSharp7 中找到。

本章代码分为以下几个主要的示例文件：

- DriveInformation
- WorkingWithFilesAndFolders
- StreamSamples
- ReaderWriterSamples
- CompressFileSample
- FileMonitor
- MemoryMappedFiles
- NamedPipes
- AnonymousPipes
- WindowsAppEditor

## 22.1 概述

当读写文件和目录时，可以使用简单的 API，也可以使用先进的 API 来提供更多的功能。还必须区分 Windows Runtime 提供的.NET 类和功能。在通用 Windows 平台(UWP)Windows 应用程序中，不能在任何目录中访问文件系统，只能访问特定的目录。或者，可以让用户选择文件。本章涵盖了所有这些选项，包括使用简单的 API 读写文件并使用流得到更多的功能；利用.NET 类型和 Windows Runtime 提供的类型，混合这两种技术以利用.NET 功能和 Windows 运行库。

使用流，也可以压缩数据，并且利用内存映射的文件和管道在不同的任务间共享数据。

## 22.2 管理文件系统

图 22-1 中的类可以用于浏览文件系统和执行操作，如移动、复制和删除文件。

这些类的作用是：

- FileSystemInfo ——这是表示任何文件系统对象的基类。
- FileInfo 和 File ——这些类表示文件系统上的文件。
- DirectoryInfo 和 Directory ——这些类表示文件系统上的文件夹。
- Path ——这个类包含的静态成员可以用于处理路径名。
- DriveInfo ——它的属性和方法提供了指定驱动器的信息。

**注意：**

目录或文件夹这两个术语经常可以互换。目录是文件系统对象的经典术语。目录包含文件和其他目录。文件夹起源于苹果的 Lisa，是一个 GUI 对象。它通常与映射到目录的图标相关联。

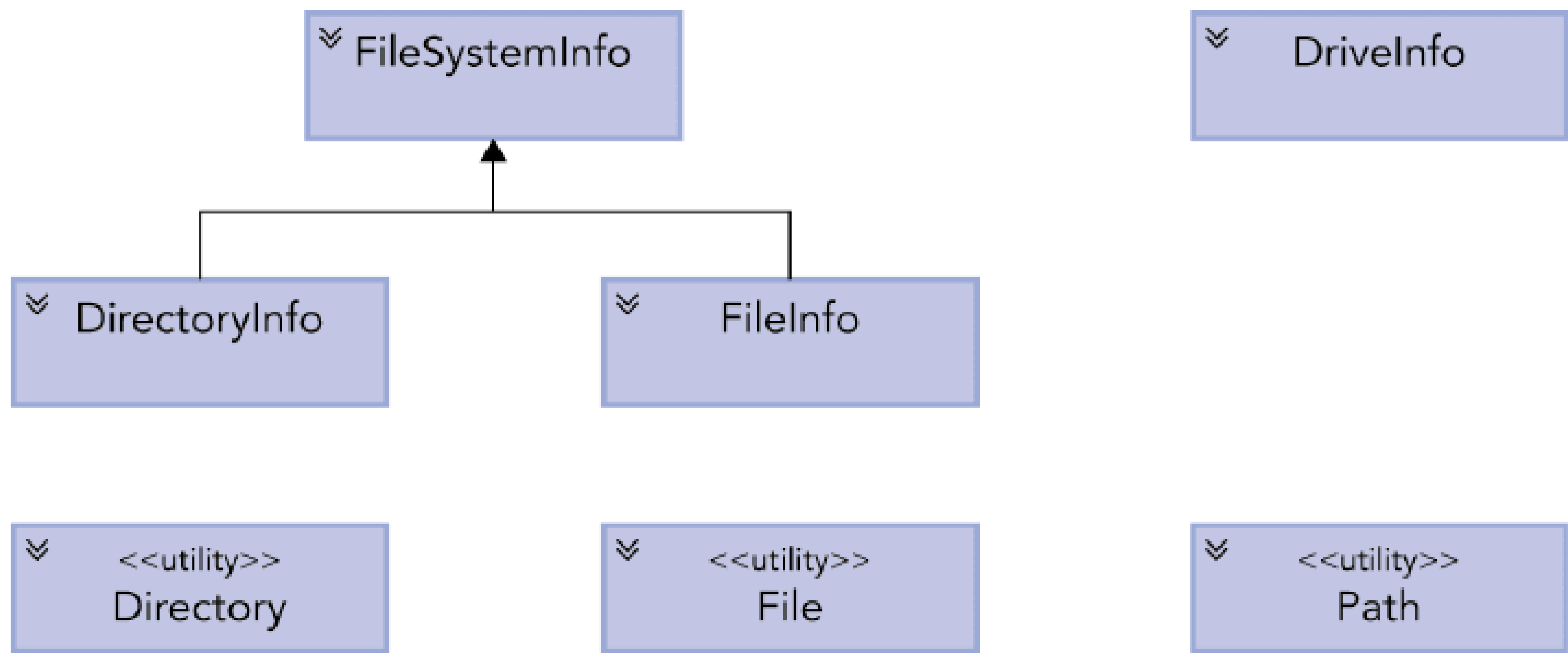



图 22-1

注意，上面的列表有两个用于表示文件夹的类，和两个用于表示文件的类。使用哪个类主要依赖于访问该文件夹或文件的次数：

- Directory 类和 File 类只包含静态方法，不能被实例化。只要调用一个成员方法，提供合适文件系统对象的路径，就可以使用这些类。如果只对文件夹或文件执行一个操作，使用这些类就很有效，因为这样可以省去创建.NET 对象的系统开销。
- DirectoryInfo 类和 FileInfo 类实现与 Directory 类和 File 类大致相同的公共方法，并拥有一些公共属性和构造函数，但它们都是有状态的，并且这些类的成员都不是静态的。需要实例化这些类，之后把每个实例与特定的文件夹或文件关联起来。如果使用同一个对象执行多个操作，使用这些类就比较有效。这是因为在构造时它们将读取合适文件系统对象的身份验证和其他信息，无论对每个对象(类实例)调用了多少方法，都不需要再次读取这些信息。比较而言，在调用每个方法时，相应的无状态类需要再次检查文件或文件夹的详细内容。

### 22.2.1 检查驱动器信息

在处理文件和目录之前，先检查驱动器信息。这使用 DriveInfo 类实现。DriveInfo 类可以扫描系统，提供可用驱动器的列表，还可以进一步提供任何驱动器的大量细节。

为了举例说明 DriveInfo 类的用法，创建一个简单的 Console 应用程序，列出计算机上的所有可用的驱动器。

DriveInformation 的示例代码使用了以下名称空间：

System

System.IO

下面的代码片段调用静态方法 DriveInfo.GetDrives。这个方法返回一个 DriveInfo 对象的数组。通过这个数组，访问每个驱动器，准备写入驱动器的名称、类型和格式信息，它还显示大小信息(代码文件 DriveInformation/Program.cs)：

```
DriveInfo[] drives = DriveInfo.GetDrives();
foreach (DriveInfo drive in drives)
{
  if (drive.IsReady)
  {
    Console.WriteLine($"Drive name: {drive.Name}");
    Console.WriteLine($"Format: {drive.DriveFormat}");
    Console.WriteLine($"Type: {drive.DriveType}");
    Console.WriteLine($"Root directory: {drive.RootDirectory}");
    Console.WriteLine($"Volume label: {drive.VolumeLabel}");
    Console.WriteLine($"Free space: {drive.TotalFreeSpace}");
    Console.WriteLine($"Available space: {drive.AvailableFreeSpace}");
    Console.WriteLine($"Total size: {drive.TotalSize}");
    Console.WriteLine();
  }
}
```

在没有 DVD 光驱、但有固态硬盘(solid-state disk，SSD)和内存卡的系统上，运行这个程序，得到如下信息：

```
Drive name: C:\
Format: NTFS
Type: Fixed
Root directory: C:\
Volume label: Windows
Free space: 289063882752
Available space: 289063882752
Total size: 509571969024

Drive name: D:\
Format: exFAT
Type: Removable
Root directory: D:\
Volume label:
Free space: 196831477760
Available space: 196831477760
Total size: 196832395264
```

在 Mac 电脑上，驱动器符是不可用的，但也可以看到 Ram 和 Network 类型，它们可以通过基于 Unix 的系统上的文件 API 来访问：

```
Drive name: /
Format: hfs
Type: Fixed
Root directory: /
Volume label: /
Free space: 170332930048
Available space: 170070786048
Total size: 249769230336

Drive name: /dev
Format: devs
Type: Ram
Root directory: /dev
Volume label: /dev
Free space: 0
Available space: 0
```

```
Total size: 184832

Drive name: /net
Format: autofs
Type: Network
Root directory: /net
Volume label: /net
Free space: 0
Available space: 0
Total size: 0
```

### 22.2.2 使用 Path 类

为了访问文件和目录，需要定义文件和目录的名称，包括父文件夹。使用字符串连接操作符合并多个文件夹和文件时，很容易遗漏单个分隔符或使用太多的字符。为此，Path 类可以提供帮助，因为这个类会添加缺少的分隔符，它还在基于 Windows 和 Unix 的系统上处理不同的平台需求。

Path 类提供了一些静态方法，可以更容易地对路径名执行操作。例如，假定要显示文件夹 D:\Projects 中 ReadMe.txt 文件的完整路径名，可以用下述代码查找文件的路径：

```
Console.WriteLine(Path.Combine(@"D:\Projects", "ReadMe.txt"));
```

Path.Combine()是这个类最常用的一个方法，Path 类还实现了其他方法，这些方法提供路径的信息，或者以要求的格式显示信息。

使用公共字段 VolumeSeparatorChar、DirectorySeparatorChar、AltDirectorySeparatorChar 和 PathSeparator，可以得到特定于平台的字符，用于分隔开硬盘、文件夹和文件，以及分隔开多个路径。在 Windows 中，这些字符是冒号(:)、反斜线(\)、正斜线(/)和分号(;)。

Path 类也帮助访问特定于用户的临时文件夹(GetTempPath)，创建临时(GetTempFileName)和随机文件名(GetRandomFileName)。注意，方法 GetTempFileName()包括文件夹，而 GetRandomFileName()只返回文件名，不包括任何文件夹。

WorkingWithFilesAndDirectories 的示例代码使用了下面的名称空间：

- System
- System.Collections.Generic
- System.IO

这个示例应用程序提供了几个命令行参数，来启动程序的不同功能。只是启动程序，没有命令行参数，或检查源代码，查看所有不同的选项。

Environment 类定义了一组特殊的文件夹。下面的代码片段通过把枚举值 SpecialFolder. MyDocuments 传递给 GetFolderPath 方法，返回 documents 文件夹 (代码文件 WorkingWithFilesAndDirectories/Program.cs)：

```
private static string GetDocumentsFolder() =>
  Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
```

Environment.SpecialFolder 是一个巨大的枚举，提供了音乐、图片、程序文件、应用程序数据，以及许多其他文件夹的值。

### 22.2.3 创建文件和文件夹

下面开始使用 File、FileInfo、Directory 和 DirectoryInfo 类。首先，方法 CreateAFile 创建文件 Sample1.txt，给文件添加字符串 Hello, World!。创建文本文件的简单方式是调用 File 类的 WriteAllText 方法。这个方法的参数是文件名和应该写入文件的字符串。一切都在一个调用中完成(代码文件 WorkingWithFilesAndDirectories/Program.cs)：

```
const string Sample1FileName = "Sample1.md";
// ...

public static void CreateAFile()
{
  string fileName = Path.Combine(GetDocumentsFolder(), Sample1FileName);
```

```
  File.WriteAllText(fileName, "Hello, World!");
}
```

要复制文件，可以使用 File 类的 Copy 方法或 FileInfo 类的 CopyTo 方法：

```
var file = new FileInfo(fileName1);
file.CopyTo(fileName2);
File.Copy(fileName1", fileName2);
```

第一个代码片段使用 FileInfo，执行的时间略长，因为需要实例化 file 对象，但是 file 已经准备好，可以在同一文件上执行进一步的操作。使用第二个例子时，不需要实例化对象来复制文件。

给构造函数传递包含对应文件系统对象的路径的字符串，就可以实例化 FileInfo 或 DirectoryInfo 类。刚才是处理文件的过程。处理文件夹的代码如下：

```
var myFolder = new DirectoryInfo(directory1);
```

如果路径代表的对象不存在，那么构建时不抛出一个异常；而是在第一次调用某个方法，实际需要相应的文件系统对象时抛出该异常。检查 Exists 属性，可以确定对象是否存在，是否具有适当的类型，这个功能由两个类实现：

```
var test = new FileInfo(@"C:\Windows");
Console.WriteLine(test.Exists);
```

请注意，这个属性要返回 true，相应的文件系统对象必须具备适当的类型。换句话说，如果实例化 FileInfo 对象时提供了文件夹的路径，或者实例化 DirectoryInfo 对象时提供了文件的路径，Exists 的值就是 false。如果有可能，这些对象的大部分属性和方法都返回一个值——它们不一定会抛出异常，仅因为调用了类型错误的对象，除非它们要求执行不可能的操作。例如，前面的代码片段可能会首先显示 false(因为 C:\Windows 是一个文件夹)，但它还显示创建文件夹的时间，因为文件夹带有该信息。然而，如果想使用 FileInfo.Open()方法打开文件夹，就好像它是一个文件那样，就会得到一个异常。

使用 FileInfo 和 DirectoryInfo 类的 MoveTo()和 Delete()方法，可以移动、删除文件或文件夹。File 和 Directory 类上的等效方法是 Move()和 Delete()。FileInfo 和 File 类也分别实现了方法 CopyTo()和 Copy()。但是，没有复制完整文件夹的方法——必须复制文件夹中的每个文件。

所有这些方法的用法都非常直观。MSDN 文档带有详细的描述。

### 22.2.4 访问和修改文件属性

下面获取有关文件的一些信息。可以使用 File 和 FileInfo 类来访问文件信息。File 类定义了静态方法，而 FileInfo 类提供了实例方法。以下代码片段展示了如何使用 FileInfo 检索多个信息。如果使用 File 类，访问速度将变慢，因为每个访问都意味着进行检查，以确定用户是否允许得到这个信息。而使用 FileInfo 类，则只有调用构造函数时才进行检查。

示例代码创建了一个新的 FileInfo 对象，并在控制台上写入属性 Name、DirectoryName、IsReadOnly、Extension、Length、CreationTime、LastAccessTime 和 Attributes 的结果(代码文件 WorkingWithFilesAndDirectories/Program.cs)：

```
private static void FileInformation(string fileName)
{
  var file = new FileInfo(fileName);
  Console.WriteLine($"Name: {file.Name}");
  Console.WriteLine($"Directory: {file.DirectoryName}");
  Console.WriteLine($"Read only: {file.IsReadOnly}");
  Console.WriteLine($"Extension: {file.Extension}");
  Console.WriteLine($"Length: {file.Length}");
  Console.WriteLine($"Creation time: {file.CreationTime:F}");
  Console.WriteLine($"Access time: {file.LastAccessTime:F}");
  Console.WriteLine($"File attributes: {file.Attributes}");
}
```

把当前目录中的 Program.cs 文件名传入这个方法：

```
FileInformation("./Program.cs");
```

在某台机器上，输出如下：

```
Name: Program.cs
Directory: C:\ProCSharpSources\files\ProfessionalCSharp7\FilesAndStreams\
FilesAndStreamsSamples\WorkingWithFilesAndDirectories
Read only: False
Extension: .cs
Length: 7637
Creation time: Friday, September 1, 2017 12:48:51 PM
Access time: Friday, September 1, 2017 4:07:12 PM
File attributes: Archive
```

不能设置 FileInfo 类的几个属性；它们只定义了 get 访问器。不能检索文件名、文件扩展名和文件的长度。可以设置创建时间和最后一次访问的时间。方法 ChangeFileProperties()向控制台写入文件的创建时间，以后把创建时间改为 2025 年的一个日期。

```
private static void ChangeFileProperties()
{
  string fileName = Path.Combine(GetDocumentsFolder(), Sample1FileName);
  var file = new FileInfo(fileName);
  if (!file.Exists)
  {
    Console.WriteLine($"Create the file {Sample1FileName} before calling this method");
    Console.WriteLine("You can do this by invoking this program with the -c argument");
    return;
  }
  Console.WriteLine($"creation time: {file.CreationTime:F}");
  file.CreationTime = new DateTime(2025, 12, 24, 15, 0, 0);
  Console.WriteLine($"creation time: {file.CreationTime:F}");
}
```

运行程序，显示文件的初始创建时间以及修改后的创建时间。将来可以用这项技术创建文件(至少可以指定创建时间)。

```
creation time: Sunday, December 20, 2015 9:41:49 AM
creation time: Wednesday, December 24, 2025 3:00:00 PM
```

**注意：**

*初看起来，能够手动修改这些属性可能很奇怪，但是它非常有用。例如，如果程序只需要读取文件、删除它，再用新内容创建一个新文件，就可以有效地修改文件，就可以通过修改创建日期来匹配旧文件的原始创建日期。*

### 22.2.5 使用 File 执行读写操作

通过 File.ReadAllText 和 File.WriteAllText，引入了一种使用字符串读写文件的方法。除了使用一个字符串之外，还可以给文件的每一行使用一个字符串。

不是把所有行读入一个字符串，而是从方法 File.ReadAllLines 中返回一个字符串数组。使用这个方法，可以对每一行执行不同的处理，但仍然需要将完整的文件读入内存(代码文件 WorkingWithFilesAndFolders/Program.cs)：

```
public static void ReadingAFileLineByLine(string fileName)
{
  string[] lines = File.ReadAllLines(fileName);
  int i = 1;
  foreach (var line in lines)
  {
    Console.WriteLine($"{i++}. {line}");
  }
  //...
}
```

要逐行读取，不需要等待所有行都读取完，可以使用方法 File.ReadLines。该方法返回 IEnumerable<string>，在读取完整个文件之前，就可以遍历它：

```
public static void ReadingAFileLineByLine(string fileName)
{
  //...
  IEnumerable<string> lines = File.ReadLines(fileName);
  i = 1;
```

```
    foreach (var line in lines)
    {
      Console.WriteLine($"{i++}. {line}");
    }
  }
```

要写入字符串集合，可以使用方法 File.WriteAllLines。该方法接受一个文件名和 IEnumerable<string>类型作为参数：

```
public static void WriteAFile()
{
  string fileName = Path.Combine(GetDocumentsFolder(), "movies.txt");
  string[] movies =
  {
    "Snow White And The Seven Dwarfs",
    "Gone With The Wind",
    "Casablanca",
    "The Bridge On The River Kwai",
    "Some Like It Hot"
  };
  File.WriteAllLines(fileName, movies);
}
```

为了把字符串追加到已有的文件中，应使用 File.AppendAllLines：

```
string[] moreMovies =
{
  "Psycho",
  "Easy Rider",
  "Star Wars",
  "The Matrix"
};
File.AppendAllLines(fileName, moreMovies);
```

## 22.3　枚举文件

处理多个文件时，可以使用 Directory 类。Directory 定义了 GetFiles()方法，它返回一个包含目录中所有文件的字符串数组。GetDirectories()方法返回一个包含所有目录的字符串数组。

所有这些方法都定义了重载方法，允许传送搜索模式和 SearchOption 枚举的一个值。SearchOption 通过使用 AllDirectories 或 TopDirectoriesOnly 值，可以遍历所有子目录，或留在顶级目录中。搜索模式不允许传递正则表达式（参见第 9 章）；它只传递简单的表达式，其中使用*表示任意字符，使用?表示单个字符。

遍历很大的目录(或子目录)时，GetFiles()和 GetDirectories()方法在返回结果之前需要完整的结果。另一种方式是使用方法 EnumerateDirectories()和 EnumerateFiles()。这些方法为搜索模式和选项提供相同的参数，但是它们使用 IEnumerable<string>立即开始返回结果。

下面是一个例子：在一个目录及其所有子目录中，删除所有以 Copy 结尾的文件，以防存在另一个具有相同名称和大小的文件。为了模拟这个操作，可以在键盘上按 Ctrl +A，选择文件夹中的所有文件，在键盘上按下 Ctrl + C，进行复制，再在鼠标仍位于该文件夹中时，在键盘上按下 Ctrl + V，粘贴文件。新文件会使用 Copy 作为后缀。

DeleteDuplicateFiles()方法迭代作为第一个参数传递的目录中的所有文件，使用选项 SearchOption.AllDirectories 遍历所有子目录。在 foreach 语句中，所迭代的当前文件与上一次迭代的文件做比较。如果文件名相同，只有 Copy 后缀不同，文件的大小也一样，就调用 FileInfo.Delete 删除复制的文件(代码文件 WorkingWithFilesAndFolders/Program.cs)：

```
private void DeleteDuplicateFiles(string directory, bool checkOnly)
{
  IEnumerable<string> fileNames = Directory.EnumerateFiles(directory,
    "*", SearchOption.AllDirectories);
  string previousFileName = string.Empty;
  foreach (string fileName in fileNames)
  {
    string previousName = Path.GetFileNameWithoutExtension(previousFileName);
    if (!string.IsNullOrEmpty(previousFileName) &&
      previousName.EndsWith("Copy") &&
```

```
      fileName.StartsWith(previousFileName.Substring(
        0, previousFileName.LastIndexOf(" - Copy"))))
    {
      var copiedFile = new FileInfo(previousFileName);
      var originalFile = new FileInfo(fileName);
      if (copiedFile.Length == originalFile.Length)
      {
        Console.WriteLine($"delete {copiedFile.FullName}");
        if (!checkOnly)
        {
          copiedFile.Delete();
        }
      }
    }
    previousFileName = fileName;
  }
}
```

## 22.4 使用流

现在，处理文件有更强大的选项：流。流的概念已经存在很长时间了。流是一个用于传输数据的对象，数据可以向两个方向传输：

- 如果数据从外部源传输到程序中，这就是读取流。
- 如果数据从程序传输到外部源中，这就是写入流。

外部源常常是一个文件，但也不完全都是文件。它还可能是：

- 使用一些网络协议读写网络上的数据，其目的是选择数据，或从另一个计算机上发送数据。
- 读写到命名管道上。
- 把数据读写到一个内存区域上。

一些流只允许写入，一些流只允许读取，一些流允许随机存取。随机存取允许在流中随机定位游标，例如，从流的开头开始读取，以后移动到流的末尾，再从流的一个中间位置继续读取。

在这些示例中，微软公司提供了一个.NET 类 System.IO.MemoryStream 对象来读写内存，而 System.Net.Sockets.NetworkStream 对象处理网络数据。Stream 类对外部数据源不做任何假定，外部数据源可以是文件流、内存流、网络流或任意数据源。

一些流也可以链接起来。例如，可以使用 DeflateStream 压缩数据。这个流可以写入 FileStream、MemoryStream 或 NetworkStream。CryptoStream 可以加密数据。也可以链接 DeflateStream 和 CryptoStream，再写入 FileStream。

**注意：**
第 24 章解释了如何使用 CryptoStream。

使用流时，外部源甚至可以是代码中的一个变量。这听起来很荒谬，但使用流在变量之间传输数据的技术是一个非常有用的技巧，可以在数据类型之间转换数据。C 语言使用类似的函数 sprintf()在整型和字符串之间转换数据类型，或者格式化字符串。

使用一个独立的对象来传输数据，比使用 FileInfo 或 DirectoryInfo 类更好，因为把传输数据的概念与特定数据源分离开来，可以更容易交换数据源。流对象本身包含许多通用代码，可以在外部数据源和代码中的变量之间移动数据，把这些代码与特定数据源的概念分离开来，就更容易实现不同环境下代码的重用。

虽然直接读写流不是那么容易，但可以使用阅读器和写入器。这是另一个关注点分离。阅读器和写入器可以读写流。例如，StringReader 和 StringWriter 类，与本章后面用于读写文本文件的两个类 StreamReader 和 StreamWriter 一样，都是同一继承树的一部分，这些类几乎一定在后台共享许多代码。在 System.IO 名称空间中，与流相关的类的层次结构如图 22-2 所示。

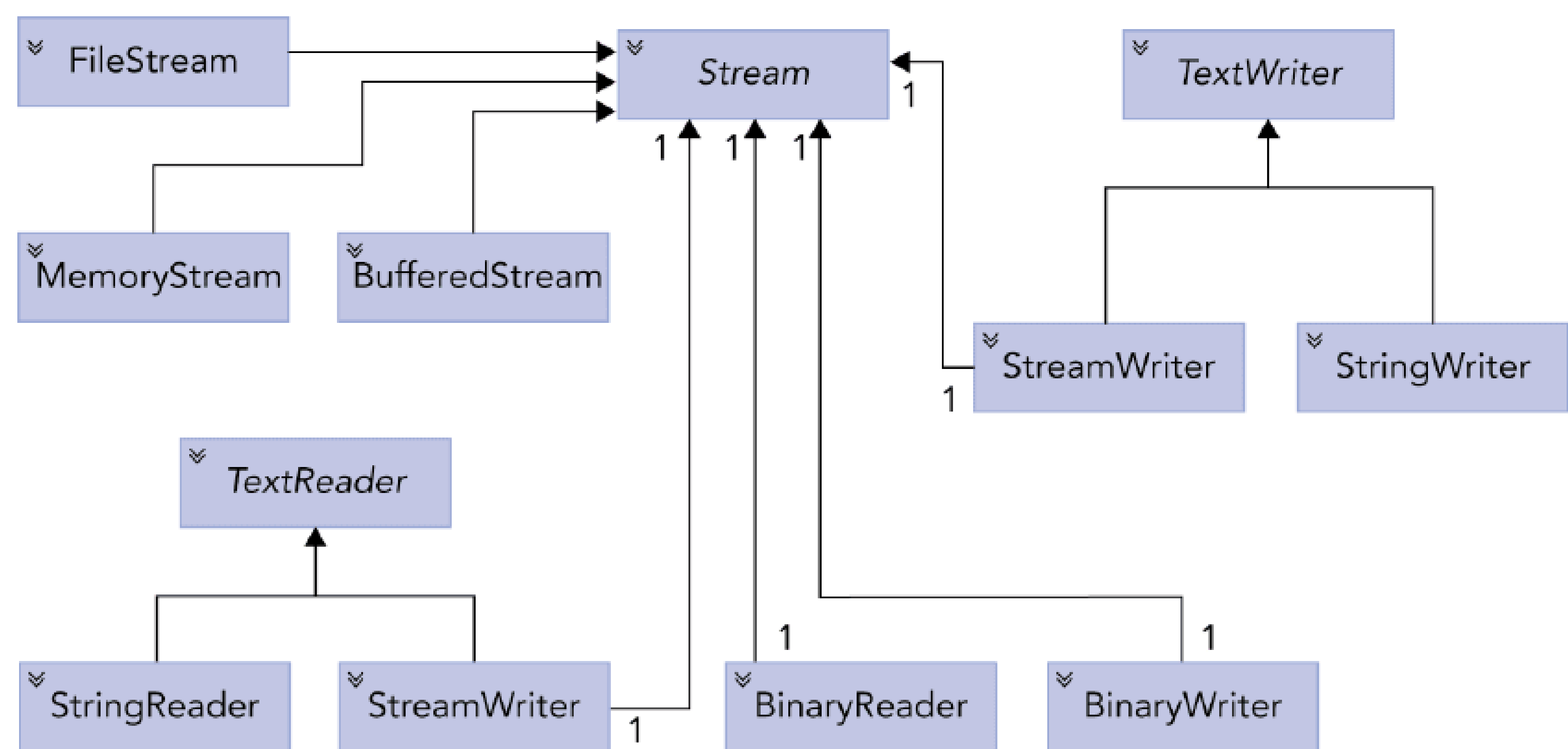



图 22-2

对于文件的读写，最常用的类如下：

- FileStream(文件流) —— 这个类主要用于在二进制文件中读写二进制数据。
- StreamReader(流读取器)和 StreamWriter(流写入器) ——这两个类专门用于读写文本格式的流产品 API。
- BinaryReader 和 BinaryWriter——这两个类专门用于读写二进制格式的流产品 API。

使用这些类和直接使用底层的流对象之间的区别是，基本流是按照字节来工作的。例如，在保存某个文档时，需要把类型为 long 的变量的内容写入一个二进制文件中，每个 long 型变量都占用 8 个字节，如果使用一般的二进制流，就必须显式地写入内存的 8 个字节中。

在 C#代码中，必须执行一些按位操作，从 long 值中提取这 8 个字节。使用 BinaryWriter 实例，可以把整个操作封装在 BinaryWriter.Write()方法的一个重载方法中，该方法的参数是 long 型，它把 8 个字节写入流中(如果流指向一个文件，就写入该文件)。对应的 BinaryReader.Read()方法则从流中提取 8 个字节，恢复 long 的值。

## 22.4.1　使用文件流

下面对流进行编程，以读写文件。FileStream 实例用于读写文件中的数据。要构造 FileStream 实例，需要以下 4 条信息：

- 要访问的文件。
- 表示如何打开文件的模式——例如，新建一个文件或打开一个现有的文件。如果打开一个现有的文件，写入操作是覆盖文件原来的内容，还是追加到文件的末尾？
- 表示访问文件的方式 —— 是只读、只写还是读写？
- 共享访问 —— 表示是否独占访问文件。如果允许其他流同时访问文件，则这些流是只读、只写还是读写文件？

第一条信息通常用一个包含文件的完整路径名的字符串来表示，本章只考虑需要该字符串的那些构造函数。除了这些构造函数外，一些其他的构造函数用本地 Windows 句柄来处理文件。其余 3 条信息分别由 3 个.NET 枚举 FileMode、FileAccess 和 FileShare 来表示，这些枚举的值很容易理解，如表 22-1 所示。

表 22-1

| 枚　举 | 值 |
|---|---|
| FileMode | Append、Create、CreateNew、Open、OpenOrCreate 或 Truncate |
| FileAccess | Read、ReadWrite 或 Write |
| FileShare | Delete、Inheritable、None、Read、ReadWrite 或 Write |

注意，对于 FileMode，如果要求的模式与文件的现有状态不一致，就会抛出一个异常。如果文件不存在，

Append、Open 和 Truncate 就会抛出一个异常；如果文件存在，CreateNew 就会抛出一个异常。Create 和 OpenOrCreate 可以处理这两种情况，但 Create 会删除任何现有的文件，新建一个空文件。因为 FileAccess 和 FileShare 枚举是按位标志，所以这些值可以与 C#的按位 OR 运算符“|”合并使用。

### 1. 创建 FileStream

StreamSamples 的示例代码使用如下名称空间：

System
System.Collections.Generic
System.Globalization
System.IO
System.Linq
System.Text
System.Threading.Tasks

FileStream 有很多构造函数。下面的示例使用带 4 个参数的构造函数(代码文件 StreamSamples/Program.cs)：

- 文件名
- FileMode 枚举值 Open，打开一个已存在的文件
- FileAccess 枚举值 Read，读取文件
- FileShare 枚举值 Read，允许其他程序读取文件，但同时不修改文件

```
private void ReadFileUsingFileStream(string fileName)
{
  const int bufferSize = 4096;
  using (var stream = new FileStream(fileName, FileMode.Open,
    FileAccess.Read, FileShare.Read))
  {
    ShowStreamInformation(stream);
    Encoding encoding = GetEncoding(stream);
    //...
```

除了使用 FileStream 类的构造函数来创建 FileStream 对象之外，还可以直接使用 File 类的 OpenRead 方法创建 FileStream。OpenRead 方法打开一个文件(类似于 FileMode.Open)，返回一个可以读取的流(FileAccess.Read)，也允许其他进程执行读取访问(FileShare.Read )：

```
using (FileStream stream = File.OpenRead(filename))
{
  //...
```

### 2. 获取流信息

Stream 类定义了属性 CanRead、CanWrite、CanSeek 和 CanTimeout,，可以读取这些属性，得到可以通过流处理的信息。为了读写流，超时值 ReadTimeout 和 WriteTimeout 指定超时，以毫秒为单位。设置这些值在网络场景中是很重要的，因为这样可以确保当读写流失败时，用户不需要等待太长时间。Position 属性返回光标在流中的当前位置。每次从流中读取一些数据，位置就移动到下一个将读取的字节上。示例代码把流的信息写到控制台上(代码文件 StreamSamples/Program.cs)：

```
private void ShowStreamInformation(Stream stream)
{
  Console.WriteLine($"stream can read: {stream.CanRead}, " +
    $"can write: {stream.CanWrite}, can seek: {stream.CanSeek}, " +
    $"can timeout: {stream.CanTimeout}");
  Console.WriteLine($"length: {stream.Length}, position: {stream.Position}");
  if (stream.CanTimeout)
  {
    Console.WriteLine($"read timeout: {stream.ReadTimeout} " +
      $"write timeout: {stream.WriteTimeout} ");
  }
}
```

对已打开的文件流运行这个程序，会得到下面的输出。位置目前为 0，因为尚未开始读取：

```
stream can read: True, can write: False, can seek: True, can timeout: False
length: 1113, position: 0
```

### 3. 分析文本文件的编码

对于文本文件，下一步是读取流中的第一个字节——序言。序言提供了文件如何编码的信息(使用的文本格式)。这也称为字节顺序标记(Byte Order Mark，BOM)。

读取一个流时，利用 ReadByte 可以从流中只读取一个字节，使用 Read()方法可以填充一个字节数组。使用 GetEncoding()方法创建了一个包含 5 字节的数组，使用 Read()方法填充字节数组。第二个和第三个参数指定字节数组中的偏移量和可用于填充的字节数。Read()方法返回读取的字节数；流可能小于缓冲区。如果没有更多的字符可用于读取，Read 方法就返回 0。

示例代码分析流的第一个字符，返回检测到的编码，并把流定位在编码字符后的位置(代码文件 StreamSamples/Program.cs)：

```
private Encoding GetEncoding(Stream stream)
{
  if (!stream.CanSeek) throw new ArgumentException(
    "require a stream that can seek");

  Encoding encoding = Encoding.ASCII;
  byte[] bom = new byte[5];
  int nRead = stream.Read(bom, offset: 0, count: 5);
  if (bom[0] == 0xff && bom[1] == 0xfe && bom[2] == 0 && bom[3] == 0)
  {
    Console.WriteLine("UTF-32");
    stream.Seek(4, SeekOrigin.Begin);
    return Encoding.UTF32;
  }
  else if (bom[0] == 0xff && bom[1] == 0xfe)
  {
    Console.WriteLine("UTF-16, little endian");
    stream.Seek(2, SeekOrigin.Begin);
    return Encoding.Unicode;
  }
  else if (bom[0] == 0xfe && bom[1] == 0xff)
  {
    Console.WriteLine("UTF-16, big endian");
    stream.Seek(2, SeekOrigin.Begin);
    return Encoding.BigEndianUnicode;
  }
  else if (bom[0] == 0xef && bom[1] == 0xbb && bom[2] == 0xbf)
  {
    Console.WriteLine("UTF-8");
    stream.Seek(3, SeekOrigin.Begin);
    return Encoding.UTF8;
  }
  stream.Seek(0, SeekOrigin.Begin);
  return encoding;
}
```

文件以 FF 和 FE 字符开头。这些字节的顺序提供了如何存储文档的信息。两字节的 Unicode 可以用小或大端字节顺序法存储。FF 紧随在 FE 之后，表示使用小端字节序，而 FE 后跟 FF，就表示使用大端字节序。这个字节顺序可以追溯到 IBM 的大型机，它使用大端字节序给字节排序，Digital Equipment 中的 PDP11 系统使用小端字节序。通过网络与采用不同字节顺序的计算机通信时，要求改变一端的字节顺序。现在，英特尔 CPU 体系结构使用小端字节序，ARM 架构允许在小端和大端字节顺序之间切换。

这些编码的其他区别是什么？在 ASCII 中，每一个字符有 7 位就足够了。ASCII 最初基于英语字母表，提供了小写字母、大写字母和控制字符。

扩展的 ASCII 利用 8 位，允许切换到特定于语言的字符。切换并不容易，因为它需要关注代码地图，也没有为一些亚洲语言提供足够的字符。UTF-16(Unicode 文本格式)解决了这个问题，它为每一个字符使用 16 位。因为对于以前的字形，UTF-16 还不够，所以 UTF-32 为每一个字符使用 32 位。虽然 Windows NT 3.1 为默认文本编码切换为 UTF-16 (在以前 ASCII 的微软扩展中)，现在最常用的文本格式是 UTF-8。在 Web 上，UTF-8 是

自 2007 年以来最常用的文本格式(这个取代了 ASCII，是以前最常见的字符编码)。UTF-8 使用可变长度的字符定义。一个字符定义为使用 1 到 6 个字节。这个字符序列在文件的开头探测 UTF-8：0xEF、0xBB、0xBF。

### 22.4.2 读取流

打开文件并创建流后，使用 Read()方法读取文件。重复此过程，直到该方法返回 0 为止。使用在前面定义的 GetEncoding()方法中创建的 Encoder，创建一个字符串。不要忘记使用 Dispose()方法关闭流。如果可能，使用 using 语句(如本代码示例所示)自动销毁流(代码文件 StreamSamples/Program.cs)：

```
public static void ReadFileUsingFileStream(string fileName)
{
  const int BUFFERSIZE = 256;
  using (var stream = new FileStream(fileName, FileMode.Open,
    FileAccess.Read, FileShare.Read))
  {
    ShowStreamInformation(stream);
    Encoding encoding = GetEncoding(stream);
    byte[] buffer = new byte[bufferSize];
    bool completed = false;
    do
    {
      int nread = stream.Read(buffer, 0, BUFFERSIZE);
      if (nread == 0) completed = true;
      if (nread < BUFFERSIZE)
      {
        Array.Clear(buffer, nread, BUFFERSIZE - nread);
      }
      string s = encoding.GetString(buffer, 0, nread);
      Console.WriteLine($"read {nread} bytes");
      Console.WriteLine(s);
    } while (!completed);
  }
}
```

### 22.4.3 写入流

把一个简单的字符串写入文本文件，就演示了如何写入流。为了创建一个可以写入的流，可以使用 File.OpenWrite()方法。这次通过 Path.GetTempFileName 创建一个临时文件名。GetTempFileName 定义的默认文件扩展名通过 Path.ChangeExtension 改为 txt(代码文件 StreamSamples/Program.cs)：

```
public static void WriteTextFile()
{
  string tempTextFileName = Path.ChangeExtension(Path.GetTempFileName(),
    "txt");
  using (FileStream stream = File.OpenWrite(tempTextFileName))
  {
    //...
```

写入 UTF-8 文件时，需要把序言写入文件。为此，可以使用 WriteByte()方法，给流发送 3 个字节的 UTF-8 序言：

```
stream.WriteByte(0xef);
stream.WriteByte(0xbb);
stream.WriteByte(0xbf);
```

这有一个替代方案。不需要记住指定编码的字节。Encoding 类已经有这些信息了。GetPreamble()方法返回一个字节数组，其中包含文件的序言。这个字节数组使用 Stream 类的 Write()方法写入：

```
byte[] preamble = Encoding.UTF8.GetPreamble();
stream.Write(preamble, 0, preamble.Length);
```

现在可以写入文件的内容。Write()方法需要写入字节数组，所以需要转换字符串。将字符串转换为 UTF-8 的字节数组，可以使用 Encoding.UTF8.GetBytes 完成这个工作，之后写入字节数组：

```
string hello = "Hello, World!";
byte[] buffer = Encoding.UTF8.GetBytes(hello);
stream.Write(buffer, 0, buffer.Length);
Console.WriteLine($"file {stream.Name} written");
```

可以使用编辑器(比如 Notepad)打开临时文件，它会使用正确的编码。

### 22.4.4　复制流

现在复制文件内容，把读写流合并起来。在下一个代码片段中，用 File.OpenRead 打开可读的流，用 File.OpenWrite 打开可写的流。使用 Stream.Read()方法读取缓冲区，用 Stream.Write()方法写入缓冲区(代码文件 StreamSamples/Program.cs)：

```
public static void CopyUsingStreams(string inputFile, string outputFile)
{
  const int BUFFERSIZE = 4096;
  using (var inputStream = File.OpenRead(inputFile))
  using (var outputStream = File.OpenWrite(outputFile))
  {
    byte[] buffer = new byte[BUFFERSIZE];
    bool completed = false;
    do
    {
      int nRead = inputStream.Read(buffer, 0, BUFFERSIZE);
      if (nRead == 0) completed = true;
      outputStream.Write(buffer, 0, nRead);
    } while (!completed);
  }
}
```

为了复制流，不需要编写读写流的代码。而可以使用 Stream 类的 CopyTo 方法，如下所示(代码文件 StreamSamples/Program.cs)：

```
public static void CopyUsingStreams2(string inputFile, string outputFile)
{
  using (var inputStream = File.OpenRead(inputFile))
  using (var outputStream = File.OpenWrite(outputFile))
  {
    inputStream.CopyTo(outputStream);
  }
}
```

### 22.4.5　随机访问流

随机访问流(甚至可以访问大文件)的一个优势是，可以快速访问文件中的特定位置。

为了了解随机存取动作，下面的代码片段创建了一个大文件。这个代码片段创建的文件 sampledata.data 包含了长度相同的记录，包括一个数字、一个文本和一个随机的日期。传递给方法的记录数通过 Enumerable.Range 方法创建。Select 方法创建了一个匿名类型，其中包含 Number、Text 和 Date 属性。除了这些记录外，还创建一个带#前缀和后缀的字符串，每个值的长度都固定，每个值之间用;作为分隔符。WriteAsync 方法将记录写入流(代码文件 StreamSamples/Program.cs)：

```
const string SampleFilePath = "./samplefile.data";
public static async Task CreateSampleFile(int nRecords)
{
  FileStream stream = File.Create(SampleFilePath);
  using (var writer = new StreamWriter(stream))
  {
    var r = new Random();
    var records = Enumerable.Range(0, nRecords).Select(x => new
    {
      Number = x,
      Text = $"Sample text {r.Next(200)}",
      Date = new DateTime(Math.Abs((long)((r.NextDouble() * 2 - 1) *
      DateTime.MaxValue.Ticks)))
    });

    foreach (var rec in records)
    {
      string date = rec.Date.ToString("d", CultureInfo.InvariantCulture);
      string s =
        $"#{rec.Number,8};{rec.Text,-20};{date}#{Environment.NewLine}";
      await writer.WriteAsync(s);
    }
```

```
    }
  }
```

**注意：**

第 17 章提到，每个实现 IDisposable 的对象都应该销毁。在前面的代码片段中，FileStream 似乎并没有销毁。然而事实并非如此。StreamWriter 销毁时，StreamWriter 会控制所使用的资源，并销毁流。为了使流打开的时间比 StreamWriter 更长，可以用 StreamWriter 的构造函数配置它。在这种情况下，需要显式地销毁流。

现在把游标定位到流中的一个随机位置，读取不同的记录。用户需要输入应该访问的记录号。流中应该访问的字节基于记录号和记录的大小。现在 Stream 类的 Seek 方法允许定位流中的光标。第二个参数指定位置是流的开头、流的末尾或是当前位置(代码文件 StreamSamples/Program.cs)：

```
public static void RandomAccessSample()
{
  try
  {
    using (FileStream stream = File.OpenRead(SampleFilePath))
    {
      byte[] buffer = new byte[RECORDSIZE];
      do
      {
        try
        {
          Console.Write("record number (or 'bye' to end): ");
          string line = Console.ReadLine();
          if (line.ToUpper().CompareTo("BYE") == 0) break;
          if (int.TryParse(line, out int record))
          {
            stream.Seek((record - 1) * RECORDSIZE, SeekOrigin.Begin);
            stream.Read(buffer, 0, RECORDSIZE);
            string s = Encoding.UTF8.GetString(buffer);
            Console.WriteLine($"record: {s}");
          }
        }
        catch (Exception ex)
        {
          Console.WriteLine(ex.Message);
        }
      } while (true);
      Console.WriteLine("finished");
    }
  }
  catch (FileNotFoundException)
  {
    Console.WriteLine("Create the sample file using the option -sample first");
  }
}
```

利用这些代码，可以尝试创建一个包含 150 万条记录或更多的文件。使用记事本打开这个大小的文件会比较慢，但是使用随机存取会非常快。根据系统、CPU 和磁盘类型，可以使用更高或更低的值来测试。

**注意：**

如果应该访问的记录的大小不固定，仍可以为大文件使用随机存取。解决这一问题的方法之一是把写入记录的位置放在文件的开头。另一个选择是读取记录所在的一个更大的块，在其中可以找到记录标识符和内存块中的记录限值条件。

### 22.4.6 使用缓存的流

从性能原因上看，在读写文件时，输出结果会被缓存。如果程序要求读取文件流中下面的两个字节，该流会把请求传递给 Windows，则 Windows 不会连接文件系统，再定位文件，并从磁盘中读取文件，仅读取两个字节。而是在一次读取过程中，检索文件中的一个大块，把该块保存在一个内存区域，即缓冲区上。以后对流中数据的请求就会从该缓冲区中读取，直到读取完该缓冲区为止。此时，Windows 会从文件中再获取另一个数据块。

写入文件的方式与此相同。对于文件，操作系统会自动完成读写操作，但需要编写一个流类，从其他没有缓存的设备中读取数据。如果是这样，就应从 BufferedStream 创建一个类，它实现一个缓冲区，并把应缓存的流传递给构造函数。但注意，BufferedStream 并不用于应用程序频繁切换读数据和写数据的情形。

# 22.5 使用读取器和写入器

使用 FileStream 类读写文本文件，需要使用字节数组，处理前一节描述的编码。有更简单的方法：使用读取器和写入器。可以使用 StreamReader 和 StreamWriter 类读写 FileStream，不需要处理字节数组和编码，比较轻松。

这是因为这些类工作的级别比较高，特别适合于读写文本。它们实现的方法可以根据流的内容，自动检测出停止读取文本较方便的位置。特别是：

- 这些类实现的方法(StreamReader.ReadLine 和 StreamWriter.WriteLine)可以一次读写一行文本。在读取文件时，流会自动确定下一个回车符的位置，并在该处停止读取。在写入文件时，流会自动把回车符和换行符追加到文本的末尾。
- 使用 StreamReader 和 StreamWriter 类，就不需要担心文件中使用的编码方式。

ReaderWriterSamples 的示例代码使用下面的名称空间：

System
System.Collections.Generic
System.Globalization
System.IO
System.Linq
System.Text
System.Threading.Tasks

## 22.5.1 StreamReader 类

先看看 StreamReader，将前面的示例转换为读取文件以使用 StreamReader。它现在看起来容易得多。StreamReader 的构造函数接收 FileStream。使用 EndOfStream 属性可以检查文件的末尾，使用 ReadLine 方法读取文本行(代码文件 ReaderWriterSamples/Program.cs)：

```
public static void ReadFileUsingReader(string fileName)
{
  var stream = new FileStream(fileName, FileMode.Open, FileAccess.Read,
    FileShare.Read);
  using (var reader = new StreamReader(stream))
  {
    while (!reader.EndOfStream)
    {
      string line = reader.ReadLine();
      Console.WriteLine(line);
    }
  }
}
```

不再需要处理字节数组和编码。然而注意，StreamReader 默认使用 UT F-8 编码。指定另一个构造函数，可以让 StreamReader 使用文件中序言定义的编码：

```
var reader = new StreamReader(stream, detectEncodingFromByteOrderMarks: true);
```

也可以显式地指定编码：

```
var reader = new StreamReader(stream, Encoding.Unicode);
```

其他构造函数允许设置要使用的缓冲区；默认为 1024 个字节。此外，还可以指定关闭读取器时不应该关闭底层流。默认情况下，关闭读取器时(使用 Dispose 方法)，会关闭底层流。

不显式实例化新的 StreamReader，而可以使用 File 类的 OpenText 方法创建 StreamReader：

```
var reader = File.OpenText(fileName);
```

对于读取文件的代码片段，该文件使用 ReadLine 方法逐行读取。StreamReader 还允许在流中使用 ReadToEnd 从光标的位置读取完整的文件：

```
string content = reader.ReadToEnd();
```

StreamReader 还允许把内容读入一个字符数组。这类似于 Stream 类的 Read 方法；它不读入字节数组，而是读入 char 数组。记住，char 类型使用两个字节。这适合于 16 位 Unicode，但不适合于 UTF-8，其中，一个字符的长度可以是 1 至 6 个字节：

```
int nChars = 100;
char[] charArray = new char[nChars];
int nCharsRead = reader.Read(charArray, 0, nChars);
```

## 22.5.2 StreamWriter 类

StreamWriter 的工作方式与 StreamReader 相同，只是 StreamWriter 仅用于写入文件(或写入另一个流)。下面的代码片段传递 FileStream，创建了一个 StreamWriter。然后把传入的字符串数组写入流(代码文件 ReaderWriterSamples/Program.cs)：

```
public static void WriteFileUsingWriter(string fileName, string[] lines)
{
  var outputStream = File.OpenWrite(fileName);
  using (var writer = new StreamWriter(outputStream))
  {
    byte[] preamble = Encoding.UTF8.GetPreamble();
    outputStream.Write(preamble, 0, preamble.Length);
    writer.Write(lines);
  }
}
```

记住，StreamWriter 默认使用 UTF-8 格式写入文本内容。通过在构造函数中设置 Encoding 对象，可以定义替代的内容。另外，类似于 StreamReader 的构造函数，StreamWriter 允许指定缓冲区的大小，以及关闭写入器时是否不应该关闭底层流。

StreamWriter 的 Write()方法定义了 17 个重载版本，允许传递字符串和一些.NET 数据类型。请记住，使用传递.NET 数据类型的方法，这些都会使用指定的编码变成字符串。要以二进制格式写入数据类型，可以使用下面介绍的 BinaryWriter。

## 22.5.3 读写二进制文件

读写二进制文件的一种选择是直接使用流类型；在这种情况下，最好使用字节数组执行读写操作。另一个选择是使用为这个场景定义的读取器和写入器：BinaryReader 和 BinaryWriter。使用它们的方式类似于使用 StreamReader 和 StreamWriter，但 BinaryReader 和 BinaryWriter 不使用任何编码。文件使用二进制格式而不是文本格式写入。

与 Stream 类型不同，BinaryWriter 为 Write()方法定义了 18 个重载版本。重载版本接受不同的类型，如下面的代码片段所示，它写入 double、int、long 和 string(代码文件 ReaderWriterSamples/Program.cs)：

```
public static void WriteFileUsingBinaryWriter(string binFile)
{
  var outputStream = File.Create(binFile);
  using (var writer = new BinaryWriter(outputStream))
  {
    double d = 47.47;
    int i = 42;
    long l = 987654321;
    string s = "sample";
    writer.Write(d);
    writer.Write(i);
    writer.Write(l);
    writer.Write(s);
```

```
    }
  }
```

为了再次读取文件，可以使用 BinaryReader。这个类定义的方法会读取所有不同的类型，如 ReadDouble、ReadInt32、ReadInt64 和 ReadString，如下所示：

```
public static void ReadFileUsingBinaryReader(string binFile)
{
  var inputStream = File.Open(binFile, FileMode.Open);
  using (var reader = new BinaryReader(inputStream))
  {
    double d = reader.ReadDouble();
    int i = reader.ReadInt32();
    long l = reader.ReadInt64();
    string s = reader.ReadString();
    Console.WriteLine($"d: {d}, i: {i}, l: {l}, s: {s}");
  }
}
```

读取文件的顺序必须完全匹配写入的顺序。创建自己的二进制格式时，需要知道存储的内容和方式，并用相应的方式读取。旧的微软 Word 文档使用二进制文件格式，而新的 docx 文件扩展是 ZIP 文件。如何读写压缩文件详见下一节。

## 22.6 压缩文件

.NET 包括使用不同的算法压缩和解压缩流的类型。可以使用 DeflateStream 和 GZipStream 压缩和解压缩流；ZipArchive 类可以创建和读取 ZIP 文件。

DeflateStream 和 GZipStream 使用相同的压缩算法(事实上，GZipStream 在后台使用 DeflateStream)，但 GZipStream 增加了循环冗余校验，来检测数据的损坏情况。Brotli 是谷歌中比较新的开源压缩算法。Brotli 的速度类似于抑制算法，但它提供了更好的压缩。与大多数其他压缩算法不同的是，它给常用的单词使用一个字典，来进行更好的压缩。目前大多数现代浏览器都支持这种算法。

使用 zip 文件的优点是可以将文件压缩到存档(使用 ZipArchive)，并且可以使用 Windows 资源管理器直接打开该存档；它自从 1998 年开始就安装到 Windows 系统中。不能使用 Windows 资源管理器打开 gzip 归档文件；打开 gzip 需要第三方工具。

**注意：**

DeflateStream 和 GZipStream 使用的算法是抑制算法。该算法由 RFC 1951 定义(https://tools.ietf.org/html/rfc1951)。这个算法被广泛认为不受专利的限制，因此得到广泛使用。

Brotli 可以在 GitHub 上的 https://github.com/google/brotli 获得，它由 RFC 7932 (https://tools.ietf.org/html/rfc7932)定义。

CompressFileSample 的示例代码使用了以下依赖项和名称空间：

**依赖项**

System.IO.Compression.Brotli

**名称空间**

System

System.Collections.Generic

System.IO

System.IO.Compression

System.Text

### 22.6.1 使用压缩流

如前所述，流的一个特性是可以将它们链接起来。为了压缩流，只需要创建 DeflateStream，并给构造函数传递另一个流(在这个例子中，是写入文件的 outputStream)，使用 CompressionMode. Compress 表示压缩。使用 Write 方法或其他功能写入这个流，如以下代码片段所示的 CopyTo()方法，就是文件压缩所需的所有操作(代码文件 CompressFileSample/Program.cs)：

```
public static void CompressFile(string fileName, string compressedFileName)
{
  using (FileStream inputStream = File.OpenRead(fileName))
  {
    FileStream outputStream = File.OpenWrite(compressedFileName);
    using (var compressStream =
      new DeflateStream(outputStream, CompressionMode.Compress))
    {
      inputStream.CopyTo(compressStream);
    }
  }
}
```

为了再次把通过 DeflateStream 压缩的文件解压缩，下面的代码片段使用 FileStream 打开文件，并创建 DeflateStream 对象，把 CompressionMode.Decompress 传入文件流，表示解压缩。Stream.CopyTo 方法把解压缩的流复制到 MemoryStream 中。然后，这个代码片段利用 StreamReader 读取 MemoryStream 中的数据，把输出写到控制台。StreamReader 配置为打开所分配的 MemoryStream (使用 leaveOpen 参数)，所以 MemoryStream 在关闭读取器后也可以使用：

```
public static void DecompressFile(string fileName)
{
  FileStream inputStream = File.OpenRead(fileName);
  using (MemoryStream outputStream = new MemoryStream())
  using (var compressStream = new DeflateStream(inputStream,
    CompressionMode.Decompress))
  {
    compressStream.CopyTo(outputStream);
    outputStream.Seek(0, SeekOrigin.Begin);
    using (var reader = new StreamReader(outputStream, Encoding.UTF8,
      detectEncodingFromByteOrderMarks: true, bufferSize: 4096,
      leaveOpen: true))
    {
      string result = reader.ReadToEnd();
      Console.WriteLine(result);
    }
    // you could use the outputStream after the StreamReader is closed
  }
}
```

### 22.6.2 使用 Brotli

使用 BrotliStream，通过 Brotli 进行压缩就像使用 deflate 一样。只需要添加 NuGet 包 System.IO.Compression.Brotli，并实例化 BrotliStream 类(代码文件 CompressFileSample/Program.cs):

```
public static void CompressFileWithBrotli(string fileName,
  string compressedFileName)
{
  using (FileStream inputStream = File.OpenRead(fileName))
  {
    FileStream outputStream = File.OpenWrite(compressedFileName);
    using (var compressStream =
      new BrotliStream(outputStream, CompressionMode.Compress))
    {
      inputStream.CopyTo(compressStream);
    }
  }
}
```

使用 BrotliStream 进行相应的解压工作：

```
public static void DecompressFileWithBrotli(string fileName)
{
```

```
    FileStream inputStream = File.OpenRead(fileName);
    using (MemoryStream outputStream = new MemoryStream())
    using (var compressStream = new BrotliStream(inputStream,
      CompressionMode.Decompress))
    {
      compressStream.CopyTo(outputStream);
      outputStream.Seek(0, SeekOrigin.Begin);
      using (var reader = new StreamReader(outputStream, Encoding.UTF8,
        detectEncodingFromByteOrderMarks: true, bufferSize: 4096,
        leaveOpen: true))
      {
        string result = reader.ReadToEnd();
        Console.WriteLine(result);
      }
    }
  }
```

### 22.6.3　压缩文件

今天，ZIP 文件格式是许多不同文件类型的标准。Word 文档(docx)以及 NuGet 包都存储为 ZIP 文件。在.NET 中，很容易创建 ZIP 归档文件。

要创建 ZIP 归档文件，可以创建一个 ZipArchive 对象。ZipArchive 包含多个 ZipArchiveEntry 对象。ZipArchive 类不是一个流，但是它使用流进行读写（类似于前面讨论的读取器和写入器）。下面的代码片段创建一个 ZipArchive，将压缩内容写入用 File.OpenWrite 打开的文件流中。添加到 ZIP 归档文件中的内容由所传递的目录定义。Directory. EnumerateFiles 枚举了目录中的所有文件，为每个文件创建一个 ZipArchiveEntry 对象。调用 Open 方法创建一个 Stream 对象。使用要读取的 Stream 的 CopyTo 方法，压缩文件，写入 ZipArchiveEntry (代码文件 CompressFileSample/Program.cs)：

```
public static void CreateZipFile(string directory, string zipFile)
{
  FileStream zipStream = File.OpenWrite(zipFile);
  using (var archive = new ZipArchive(zipStream, ZipArchiveMode.Create))
  {
    IEnumerable<string> files = Directory.EnumerateFiles(
      directory, "*", SearchOption.TopDirectoryOnly);
    foreach (var file in files)
    {
      ZipArchiveEntry entry = archive.CreateEntry(Path.GetFileName(file));
      using (FileStream inputStream = File.OpenRead(file))
      using (Stream outputStream = entry.Open())
      {
        inputStream.CopyTo(outputStream);
      }
    }
  }
}
```

## 22.7　观察文件的更改

使用 FileSystemWatcher 可以监视文件的更改。事件在创建、重命名、删除和更改文件时触发。这可用于如下场景：需要对文件的变更做出反应，例如，服务器上传文件时，或文件缓存在内存中，而缓存需要在文件更改时失效。

因为 FileSystemWatcher 易于使用，所以下面直接开始一个示例。FileMonitor 的示例代码利用以下名称空间：

System

System.IO

示例代码在 WatchFiles()方法中开始观察文件。使用 FileSystemWatcher 的构造函数时，可以提供应该观察的目录。还可以提供一个过滤器，只过滤出与过滤表达式匹配的特定文件。当设置属性 IncludeSubdirectories 时，可以定义是否应该只观察指定目录中的文件，或者是否还应该观察子目录中的文件。对于 Created、Changed、Deleted 和 Renamed 事件，提供事件处理程序。所有这些事件的类型都是 FileSystemEventHandler，只有 Renamed 事件的类型是 RenamedEventHandler。RenamedEventHandler 派生自 FileSystemEventHandler，提供了事件的附加

信息(代码文件 FileMonitor/Program.cs)：

```
private static FileSystemWatcher s_watcher;

public static void WatchFiles(string path, string filter)
{
  s_watcher = new FileSystemWatcher(path, filter)
  {
    IncludeSubdirectories = true
  };
  s_watcher.Created += OnFileChanged;
  s_watcher.Changed += OnFileChanged;
  s_watcher.Deleted += OnFileChanged;
  s_watcher.Renamed += OnFileRenamed;
  s_watcher.EnableRaisingEvents = true;
  Console.WriteLine("watching file changes...");
}
```

因文件变更而接收到的信息是 FileSystemEventArgs 类型。它包含了变更文件的名字，这种变更是一个 WatcherChangeTypes 类型的枚举：

```
private static void OnFileChanged(object sender, FileSystemEventArgs e)
{
  Console.WriteLine($"file {e.Name} {e.ChangeType}");
}
```

重命名文件时，通过 RenamedEventArgs 参数收到其他信息。这个类型派生自 FileSystemEventArgs，它定义了文件原始名称的额外信息：

```
private static void OnFileRenamed(object sender, RenamedEventArgs e)
{
  Console.WriteLine($"file {e.OldName} {e.ChangeType} to {e.Name}");
}
```

指定要观察的文件夹和*.txt 作为过滤器，启动应用程序，创建文件 sample1.txt，添加内容，把它重命名为 sample2. txt，最后删除它，输出如下。

```
watching file changes...
file New Text Document.txt Created
file New Text Document.txt Renamed to sample1.txt
file sample1.txt Changed
file sample1.txt Changed
file sample1.txt Renamed to sample2.txt
file sample2.txt Deleted
```

## 22.8 使用内存映射的文件

内存映射文件允许访问文件，或在不同的进程中共享内存。这个技术有几个场景和特点：

- 使用文件地图，快速随机访问大文件
- 在不同的进程或任务之间共享文件
- 在不同的进程或任务之间共享内存
- 使用访问器直接从内存位置进行读写
- 使用流进行读写

内存映射文件 API 允许使用物理文件或共享的内存，其中把系统的页面文件用作后备存储器。共享的内存可以大于可用的物理内存，所以需要一个后备存储器。可以为特定的文件或共享的内存创建一个内存映射文件。使用这两个选项，可以给内存映射指定名称。使用名称，允许不同的进程访问同一个共享的内存。

创建了内存映射之后，就可以创建一个视图。视图用于映射完整内存映射文件的一部分，以访问它，进行读写。

MemoryMappedFilesSample 利用下面的名称空间：

System
System.IO
System.IO.MemoryMappedFiles

System.Threading

System.Threading.Tasks

示例应用程序演示了如何通过内存映射文件，使用这两种视图访问器和流完成多个任务。一个任务是创建内存映射文件和写入数据；另一个任务是读取数据。

> **注意：**
> 示例代码使用了任务和事件。任务和任务之间的同步参见第 21 章。

准备好映射，写入数据时，需要一些基础设施来创建任务，发出信号。映射的名称和 ManualResetEventSlim 对象定义为 Program 类的一个成员(代码文件 MemoryMappedFilesSample/Program.cs)：

```
private ManualResetEventSlim _mapCreated =
  new ManualResetEventSlim(initialState: false);

private ManualResetEventSlim _dataWrittenEvent =
  new ManualResetEventSlim(initialState: false);

private const string MAPNAME = "SampleMap";
```

在 Main()方法中使用 Task.Run()方法开始执行任务：

```
public void Run()
{
  Task.Run(() => WriterAsync());
  Task.Run(() => Reader());
  Console.WriteLine("tasks started");
  Console.ReadLine();
}
```

现在使用访问器创建读取器和写入器。

## 22.8.1　使用访问器创建内存映射文件

为了创建一个基于内存的内存映射文件，写入器调用了 MemoryMappedFile.CreateOrOpen 方法。这个方法打开第一个参数指定名称的对象，如果它不存在，就创建一个新对象。要打开现有的文件，可以使用 OpenExisting 方法。为了访问物理文件，可以使用 CreateFromFile 方法。

示例代码中使用的其他参数是内存映射文件的大小和所需的访问。创建内存映射文件后，给事件 _mapCreated 发出信号，给其他任务提供信息，说明已经创建了内存映射文件，可以打开它了。调用方法 CreateViewAccessor，返回一个 MemoryMappedViewAccessor，以访问共享的内存。使用视图访问器，可以定义这一任务使用的偏移量和大小。当然，可以使用的最大大小是内存映射文件的大小。这个视图用于写入，因此文件访问设置为 MemoryMappedFileAccess.Write。

接下来，使用 MemoryMappedViewAccessor 的重载 Write 方法，可以将原始数据类型写入共享内存。Write 方法总是需要位置信息，来指定数据应该写入的位置。写入所有的数据之后，给一个事件发出信号，通知读取器，现在可以开始读取了(代码文件 MemoryMappedFilesSample/Program.cs):

```
private async Task WriterAsync()
{
  try
  {
    using (MemoryMappedFile mappedFile = MemoryMappedFile.CreateOrOpen(
      MAPNAME, 10000, MemoryMappedFileAccess.ReadWrite))
    {
      _mapCreated.Set(); // signal shared memory segment created
      Console.WriteLine("shared memory segment created");
      using (MemoryMappedViewAccessor accessor = mappedFile.CreateViewAccessor(
        0, 10000, MemoryMappedFileAccess.Write))
      {
        for (int i = 0, pos = 0; i < 100; i++, pos += 4)
        {
          accessor.Write(pos, i);
          Console.WriteLine($"written {i} at position {pos}");
          await Task.Delay(10);
        }
```

```
          _dataWrittenEvent.Set(); // signal all data written
          Console.WriteLine("data written");
        }
      }
    }
    catch (Exception ex)
    {
      Console.WriteLine($"writer {ex.Message}");
    }
  }
```

读取器首先等待创建内存映射文件，再使用 MemoryMappedFile.OpenExisting 打开它。读取器只需要映射的读取权限。之后，与前面的写入器类似，创建一个视图访问器。在读取数据之前，等待设置_dataWrittenEvent。读取类似于写入，因为也要提供应该访问数据的位置，但是不同的 Read 方法，如 ReadInt32，用于读取不同的数据类型：

```
  private void Reader()
  {
    try
    {
      Console.WriteLine("reader");
      _mapCreated.Wait();
      Console.WriteLine("reader starting");
      using (MemoryMappedFile mappedFile = MemoryMappedFile.OpenExisting(
        MAPNAME, MemoryMappedFileRights.Read))
      {
        using (MemoryMappedViewAccessor accessor = mappedFile.CreateViewAccessor(
          0, 10000, MemoryMappedFileAccess.Read))
        {
          _dataWrittenEvent.Wait();
          Console.WriteLine("reading can start now");
          for (int i = 0; i < 400; i += 4)
          {
            int result = accessor.ReadInt32(i);
            Console.WriteLine($"reading {result} from position {i}");
          }
        }
      }
    }
    catch (Exception ex)
    {
      Console.WriteLine($"reader {ex.Message}");
    }
  }
```

运行应用程序，输出如下：

```
reader
reader starting
tasks started
shared memory segment created
written 0 at position 0
written 1 at position 4
written 2 at position 8
...
written 99 at 396
data written
reading can start now
reading 0 from position 0
reading 1 from position 4
...
```

### 22.8.2 使用流创建内存映射文件

除了用内存映射文件写入原始数据类型之外，还可以使用流。流允许使用读取器和写入器，如本章前面所述。现在创建一个写入器来使用 StreamWriter。MemoryMappedFile 中的方法 CreateViewStream()返回 MemoryMappedViewStream。这个方法非常类似于前面使用的 CreateViewAccessor()方法，也是在映射内定义一个视图，有了偏移量和大小，可以方便地使用流的所有特性。

然后使用 WriteLineAsync()方法把一个字符串写到流中。StreamWriter 缓存写入操作，所以流的位置不是在每个写入操作中都更新，只在写入器写入块时才更新。为了用每次写入的内容刷新缓存，要把 StreamWriter 的

AutoFlush 属性设置为 true(代码文件 MemoryMappedFilesSample/Program.cs):

```
private async Task WriterUsingStreams()
{
  try
  {
    using (MemoryMappedFile mappedFile = MemoryMappedFile.CreateOrOpen(
      MAPNAME, 10000, MemoryMappedFileAccess.ReadWrite))
    {
      _mapCreated.Set(); // signal shared memory segment created
      Console.WriteLine("shared memory segment created");
      MemoryMappedViewStream stream = mappedFile.CreateViewStream(
        0, 10000, MemoryMappedFileAccess.Write);
      using (var writer = new StreamWriter(stream))
      {
        writer.AutoFlush = true;
        for (int i = 0; i < 100; i++)
        {
          string s = $"some data {i}";
          Console.WriteLine($"writing {s} at {stream.Position}");
          await writer.WriteLineAsync(s);
        }
      }
      _dataWrittenEvent.Set(); // signal all data written
      Console.WriteLine("data written");
    }
  }
  catch (Exception ex)
  {
    Console.WriteLine($"writer {ex.Message}");
  }
}
```

读取器同样用 CreateViewStream 创建了一个映射视图流，但这次需要读取权限。现在可以使用 StreamReader()方法从共享内存中读取内容：

```
private async Task ReaderUsingStreams()
{
  try
  {
    Console.WriteLine("reader");
    _mapCreated.Wait();
    Console.WriteLine("reader starting");
    using (MemoryMappedFile mappedFile = MemoryMappedFile.OpenExisting(
      MAPNAME, MemoryMappedFileRights.Read))
    {
      MemoryMappedViewStream stream = mappedFile.CreateViewStream(
        0, 10000, MemoryMappedFileAccess.Read);
      using (var reader = new StreamReader(stream))
      {
        _dataWrittenEvent.Wait();
        Console.WriteLine("reading can start now");
        for (int i = 0; i < 100; i++)
        {
          long pos = stream.Position;
          string s = await reader.ReadLineAsync();
          Console.WriteLine($"read {s} from {pos}");
        }
      }
    }
  }
  catch (Exception ex)
  {
    Console.WriteLine($"reader {ex.Message}");
  }
}
```

运行应用程序时，可以看到读写的数据。写入数据时，流中的位置总是更新，因为设置了 AutoFlush 属性。读取数据时，总是读取 1024 字节的块。

```
tasks started
reader
reader starting
shared memory segment created
writing some data 0 at 0
writing some data 1 at 13
```

```
writing some data 2 at 26
writing some data 3 at 39
writing some data 4 at 52
...
data written
reading can start now
read some data 0 from 0
read some data 1 from 1024
read some data 2 from 1024
read some data 3 from 1024
...
```

通过内存映射文件通信时，必须同步读取器和写入器，这样读取器才知道数据何时可用。下一节讨论的管道给这样的场景提供了其他选项。

# 22.9 使用管道通信

为了在线程和进程之间通信，在不同的系统之间快速通信，可以使用管道。在.NET 中，管道实现为流，因此不仅可以把字节发送到管道，还可以使用流的所有特性，如读取器和写入器。

管道实现为不同的类型：一种是命名管道，其中的名称可用于连接到每一端，另一种是匿名管道。匿名管道不能用于不同系统之间的通信；只能用于一个父子进程之间的通信或不同任务之间的通信。

所有管道示例的代码都利用以下名称空间：

System

System.IO

System.IO.Pipes

System.Threading

System.Threading.Tasks

下面先使用命名管道在不同的进程之间进行通信。在第一个示例应用程序中，使用了两个控制台应用程序。一个充当服务器，从管道中读取数据；另一个把消息写入管道。

## 22.9.1 创建命名管道服务器

通过创建 NamedPipeServerStream 的一个新实例，来创建服务器。NamedPipeServerStream 派生自基类 PipeStream，PipeStream 派生自 Stream 基类，因此可以使用流的所有功能，例如，可以创建 CryptoStream 或 GZipStream，把加密或压缩的数据写入命名管道。构造函数需要管道的名称，通过管道通信的多个进程可以使用该管道。

用于下面代码片段的第二个参数定义了管道的方向。服务器流用于读取，因此将方向设置为 PipeDirection.In。命名管道也可以是双向的，用于读写，此时使用 PipeDirection.InOut。匿名管道只能是单向的。接下来，调用 WaitForConnection()方法，命名管道等待写入方的连接。然后，在一个循环中(直到收到消息“bye”)，管道服务器把消息读入缓冲区数组，把消息写到控制台(代码文件 PipesReader/Program.cs):

```
private static void PipesReader(string pipeName)
{
  try
  {
    using (var pipeReader =
      new NamedPipeServerStream(pipeName, PipeDirection.In))
    {
      pipeReader.WaitForConnection();
      Console.WriteLine("reader connected");
      const int BUFFERSIZE = 256;
      bool completed = false;
      while (!completed)
      {
        byte[] buffer = new byte[BUFFERSIZE];
        int nRead = pipeReader.Read(buffer, 0, BUFFERSIZE);
        string line = Encoding.UTF8.GetString(buffer, 0, nRead);
        Console.WriteLine(line);
```

```
        if (line == "bye") completed = true;
      }
    }
    Console.WriteLine("completed reading");
    Console.ReadLine();
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

以下是可以用命名管道配置的其他一些选项：

- 可以把枚举 PipeTransmissionMode 设置为 Byte 或 Message。设置为 Byte，就发送一个连续的流，设置为 Message，就可以检索每条消息。
- 使用管道选项，可以指定 WriteThrough 立即写入管道，而不缓存。
- 可以为输入和输出配置缓冲区大小。
- 配置管道安全性，指定谁允许读写管道。安全性参见第 24 章。
- 可以配置管道句柄的可继承性，这对与子进程进行通信是很重要的。

因为 NamedPipeServerStream 是一个流，所以可以使用 StreamReader，而不是读取字节数组，该方法简化了代码：

```
var pipeReader = new NamedPipeServerStream(pipeName, PipeDirection.In);
using (var reader = new StreamReader(pipeReader))
{
  pipeReader.WaitForConnection();
  Console.WriteLine("reader connected");
  bool completed = false;
  while (!completed)
  {
    string line = reader.ReadLine();
    Console.WriteLine(line);
    if (line == "bye") completed = true;
  }
}
```

## 22.9.2　创建命名管道客户端

现在需要一个客户端。服务器读取消息，客户端就写入它们。

通过实例化一个 NamedPipeClientStream 对象来创建客户端。因为命名管道可以在网络上通信，所以需要服务器名称、管道的名称和管道的方向。客户端通过调用 Connect()方法来连接。连接成功后，就在 StreamWriter 上调用 WriteLine，把消息发送给服务器。默认情况下，消息不立即发送，而是缓存起来。调用 Flush()方法把消息推到服务器上。也可以立即传送所有的消息，而不调用 Flush()方法。为此，必须配置选项，在创建管道时遍历缓存文件(代码文件 PipesWriter/Program.cs)：

```
public static void PipesWriter(string serverName, string pipeName)
{
  var pipeWriter = new NamedPipeClientStream(serverName,
    pipeName, PipeDirection.Out);
    using (var writer = new StreamWriter(pipeWriter))
    {
      pipeWriter.Connect();
      Console.WriteLine("writer connected");
      bool completed = false;
      while (!completed)
      {
        string input = ReadLine();
        if (input == "bye") completed = true;
        writer.WriteLine(input);
        writer.Flush();
      }
    }
    Console.WriteLine("completed writing");
  }
```

为了在 Visual Studio 内开始两个项目，可以用 Project | Set Startup Projects 配置多个启动项目。运行应用程

序，一个控制台的输入就在另一个控制台上回应。

### 22.9.3 创建匿名管道

下面对匿名管道执行类似的操作。通过匿名管道，创建两个彼此通信的任务。为了给管道的创建发出信号，使用 ManualResetEventSlim 对象，与内存映射文件一样。在 Program 类的 Run 方法中，创建两个任务，调用 Reader 和 Writer 方法(代码文件 AnonymousPipes/Program.cs)：

```
private string _pipeHandle;
private ManualResetEventSlim _pipeHandleSet;

static void Main()
{
  var p = new Program();
  p.Run();
  Console.ReadLine();
}

public void Run()
{
  _pipeHandleSet = new ManualResetEventSlim(initialState: false);
  Task.Run(() => Reader());
  Task.Run(() => Writer());
  Console.ReadLine();
}
```

创建一个 AnonymousPipeServerStream，定义 PipeDirection.In，把服务器端充当读取器。通信的另一端需要知道管道的客户端句柄。这个句柄在 GetClientHandleAsString 方法中转换为一个字符串，赋予_pipeHandle 变量。这个变量以后由充当写入器的客户端使用。在最初的处理后，管道服务器可以作为一个流，因为它本来就是一个流：

```
private void Reader()
{
  try
  {
    var pipeReader = new AnonymousPipeServerStream(PipeDirection.In,
      HandleInheritability.None);
    using (var reader = new StreamReader(pipeReader))
    {
      _pipeHandle = pipeReader.GetClientHandleAsString();
      Console.WriteLine($"pipe handle: {_pipeHandle}");
      _pipeHandleSet.Set();
      bool end = false;
      while (!end)
      {
        string line = reader.ReadLine();
        Console.WriteLine(line);
        if (line == "end") end = true;
      }
      Console.WriteLine("finished reading");
    }
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

客户端代码等到变量_pipeHandleSet 发出信号，就打开由_pipeHandle 变量引用的管道句柄。后来的处理用 StreamWriter 继续：

```
private void Writer()
{
  Console.WriteLine("anonymous pipe writer");
  _pipeHandleSet.Wait();
  var pipeWriter = new AnonymousPipeClientStream(
    PipeDirection.Out, _pipeHandle);
  using (var writer = new StreamWriter(pipeWriter))
  {
    writer.AutoFlush = true;
    Console.WriteLine("starting writer");
```

```
        for (int i = 0; i < 5; i++)
        {
          writer.WriteLine($"Message {i}");
          Task.Delay(500).Wait();
        }
        writer.WriteLine("end");
      }
    }
```

运行应用程序时，两个任务就相互通信，在任务之间发送数据。

# 22.10　通过 Windows 运行库使用文件和流

通过 Windows 运行库，可以使用本地类型实现流。尽管它们用本地代码实现，但看起来类似于.NET 类型。然而，它们是有区别的：对于流，Windows 运行库在名称空间 Windows.Storage.Streams 中实现自己的类型。其中包含 FileInputStream、FileOutputStream 和 RandomAccessStreams 等类。所有这些类都基于接口，例如 IInputStream、IOutputStream 和 IRandomAccessStream。还有读取器和写入器的概念。Windows 运行库的读取器和写入器类型是 DataReader 和 DataWriter。

下面看看它与前面的.NET 流有什么不同，.NET 流和类型如何映射到这些本地类型上。

## 22.10.1　Windows App 编辑器

下面使用 Blank App (Universal Windows) Visual Studio 模板创建一个编辑器。

为了添加打开和保存文件的命令，在主页上添加一个带 AppBarButton 元素的 CommandBar (代码文件 WindowsAppEditor/MainPage. xaml)：

```
<Page.BottomAppBar>
  <CommandBar IsOpen="True">
    <AppBarButton Icon="OpenFile" Label="Open" Click="{x:Bind OnOpen}" />
    <AppBarButton Icon="Save" Label="Save" Click="{x:Bind OnSave}" />
  </CommandBar>
</Page.BottomAppBar>
```

添加到 Grid 中的 TextBox 接收文件的内容：

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
  <TextBox x:Name="text1" HorizontalTextAlignment="Left"
    AcceptsReturn="True" />
</Grid>
```

OnOpen 句柄首先启动对话框，用户可以在其中选择文件。记住，前面使用了 OpenFileDialog。在 Windows 应用程序中，可以使用选择器。要打开文件，FileOpenPicker 是首选的类型。可以配置此选择器，为用户定义建议的开始位置。将 SuggestedStartLocation 设置为 PickerLocationId. DocumentsLibrary，打开用户的文档文件夹。PickerLocationId 是定义各种特殊文件夹的枚举。

接下来，FileTypeFilter 集合指定应该为用户列出的文件类型。最后，方法 PickSingleFileAsync 返回用户选择的文件。为了让用户选择多个文件，可以使用方法 PickMultipleFilesAsync。这个方法返回一个 StorageFile。StorageFile 是在 Windows.Storage 名称空间中定义的。这个类相当于 FileInfo 类，用于打开、创建、复制、移动和删除文件(代码文件 WindowsAppEditor/MainPage.xaml.cs)：

```
public async void OnOpen()
{
  try
  {
    var picker = new FileOpenPicker()
    {
      ViewMode = PickerViewMode.Thumbnail,
      SuggestedStartLocation = PickerLocationId.DocumentsLibrary
    };
    picker.FileTypeFilter.Add(".txt");
    picker.FileTypeFilter.Add(".md");

    StorageFile file = await picker.PickSingleFileAsync();
    //...
```

现在，使用方法 OpenReadAsync()打开文件。这个方法返回一个实现了接口 IRandomAccessStream-WithContentType 的流，IRandomAccessStreamWithContentType 派生于接口 IRandomAccessStream、IInputStream、IOuputStream、IContentProvider 和 IDisposable。IRandomAccessStream 允许使用 Seek 方法随机访问流，提供了流大小的信息。IInputStream 定义了读取流的方法 ReadAsync。IOutputStream 正好相反，它定义了 WriteAsync 和 FlushAsync 方法。IContentTypeProvider 定义了属性 ContentType，提供文件内容的信息。还记得文本文件的编码吗？现在可以调用 ReadAsync()方法，读取流的内容。然而，Windows 运行库也知道前面讨论的读取器和写入器概念。DataReader 通过构造函数接受 IInputStream。DataReader 类型定义的方法可以读取原始数据类型，如 ReadInt16、ReadInt32 和 ReadDateTime。使用 ReadBytes 可以读取字节数组，使用 ReadString 可以读取字符串。ReadString()方法需要要读取的字符数。字符串赋给 TextBox 控件的 Text 属性，来显示内容：

```
//...
  if (file != null)
  {
    IRandomAccessStreamWithContentType stream = await file.OpenReadAsync();
    using (var reader = new DataReader(stream))
    {
      await reader.LoadAsync((uint)stream.Size);
      text1.Text = reader.ReadString((uint)stream.Size);
    }
  }
}
catch (Exception ex)
{
  var dlg = new MessageDialog(ex.Message, "Error");
  await dlg.ShowAsync();
}
```

**注意：**

与.NET Framework 的读取器和写入器类似，DataReader 和 DataWriter 管理通过构造函数传递的流。在销毁读取器和写入器时，流也会销毁。在.NET 类中，为了底层流打开更长时间，可以在构造函数中设置 leaveOpen 参数。对于 Windows 运行库类型，可以调用方法 DetachStream，把读取器和写入器与流分离开。

保存文档时，调用 OnSave()方法。首先，FileSavePicker 用于允许用户选择文档，与 FileOpenPicker 类似。接下来，使用 OpenTransactedWriteAsync 打开文件。NTFS 文件系统支持事务；这些都不包含在.NET Framework 中，但可用于 Windows 运行库。OpenTransactedWriteAsync 返回一个实现了接口 IStorageStreamTransaction 的 StorageStreamTransaction 对象。这个对象本身并不是流，但是它包含了一个可以用 Stream 属性引用的流。这个属性返回一个 IRandomAccessStream 流。与创建 DataReader 类似，可以创建一个 DataWriter，写入原始数据类型，包括字符串，如这个例子所示。StoreAsync 方法最后把缓冲区的内容写到流中。销毁写入器之前，需要调用 CommitAsync 方法来提交事务：

```
public async void OnSave()
{
  try
  {
    var picker = new FileSavePicker()
    {
      SuggestedStartLocation = PickerLocationId.DocumentsLibrary,
      SuggestedFileName = "New Document"
    };
    picker.FileTypeChoices.Add("Plain Text", new List<string>() { ".txt" });
    StorageFile file = await picker.PickSaveFileAsync();
    if (file != null)
    {
      using (StorageStreamTransaction tx =
        await file.OpenTransactedWriteAsync())
      {
        IRandomAccessStream stream = tx.Stream;
        stream.Seek(0);
        using (var writer = new DataWriter(stream))
        {
          writer.WriteString(text1.Text);
          tx.Stream.Size = await writer.StoreAsync();
```

```
                await tx.CommitAsync();
            }
          }
        }
      }
      catch (Exception ex)
      {
        var dlg = new MessageDialog(ex.Message, "Error");
        await dlg.ShowAsync();
      }
    }
```

DataWriter 不把定义 Unicode 文件种类的序言添加到流中。需要明确这么做，如本章前面所述。DataWriter 只通过设置 UnicodeEncoding 和 ByteOrder 属性来处理文件的编码。默认设置是 UnicodeEncoding.Utf8 和 ByteOrder.BigEndian。除了使用 DataWriter 之外，还可以利用 StreamReader 和 StreamWriter 以及.NET Stream 类的功能，见下一节。

## 22.10.2 把 Windows Runtime 类型映射为.NET 类型

下面开始读取文件。为了把 Windows Runtime 流转换为.NET 流用于读取，可以使用扩展方法 AsStreamForRead。这个方法是在程序集 System.Runtime.WindowsRuntime 的 System.IO 名称空间中定义(必须打开)。这个方法创建了一个新的 Stream 对象，来管理 IInputStream。现在，可以使用它作为正常的.NET 流，如前所述，例如，给它传递一个 StreamReader，使用这个读取器访问文件：

```
public async void OnOpenDotnet()
{
  try
  {
    var picker = new FileOpenPicker()
    {
      ViewMode = PickerViewMode.Thumbnail,
      SuggestedStartLocation = PickerLocationId.DocumentsLibrary
    };
    picker.FileTypeFilter.Add(".txt");
    picker.FileTypeFilter.Add(".md");

    StorageFile file = await picker.PickSingleFileAsync();
    if (file != null)
    {
      IRandomAccessStreamWithContentType wrtStream =
        await file.OpenReadAsync();
      Stream stream = wrtStream.AsStreamForRead();
      using (var reader = new StreamReader(stream))
      {
        text1.Text = await reader.ReadToEndAsync();
      }
    }
  }
  catch (Exception ex)
  {
    var dlg = new MessageDialog(ex.Message, "Error");
    await dlg.ShowAsync();
  }
}
```

所有的 Windows Runtime 流类型都很容易转换为.NET 流，反之亦然。表 22-2 列出了所需的方法：

表 22-2

| 从 | 转 换 为 | 方　法 |
|---|---|---|
| IRandomAccessStream | Stream | AsStream |
| IInputStream | Stream | AsStreamForRead |
| IOutputStream | Stream | AsStreamForWrite |
| Stream | IInputStream | AsInputStream |
| Stream | IOutputStream | AsOutputStream |
| Stream | IRandomAccessStream | AsRandomAccessStream |

现在将更改保存到文件中。用于写入的流通过扩展方法 AsStreamForWrite 转换。现在，这个流可以使用 StreamWriter 类写入。代码片段也把 UFT- 8 编码的序言写入文件：

```
public async void OnSaveDotnet()
{
  try
  {
    var picker = new FileSavePicker()
    {
      SuggestedStartLocation = PickerLocationId.DocumentsLibrary,
      SuggestedFileName = "New Document"
    };
    picker.FileTypeChoices.Add("Plain Text", new List<string>() { ".txt" });
    StorageFile file = await picker.PickSaveFileAsync();
    if (file != null)
    {
      StorageStreamTransaction tx = await file.OpenTransactedWriteAsync();
      using (var writer = new StreamWriter(tx.Stream.AsStreamForWrite()))
      {
        byte[] preamble = Encoding.UTF8.GetPreamble();
        await stream.WriteAsync(preamble, 0, preamble.Length);
        await writer.WriteAsync(text1.Text);
        await writer.FlushAsync();
        tx.Stream.Size = (ulong)stream.Length;
        await tx.CommitAsync();
      }
    }
  }
  catch (Exception ex)
  {
    var dlg = new MessageDialog(ex.Message, "Error");
    await dlg.ShowAsync();
  }
}
```

## 22.11 小结

本章介绍了如何在 C#代码中使用.NET 基类来访问文件系统。在这两种情况下，基类提供的对象模型比较简单，但功能强大，从而很容易执行这些领域中几乎所有的操作。对于文件系统，可以复制文件；移动、创建、删除文件和文件夹；读写二进制文件和文本文件。

本章学习了如何使用压缩算法和 ZIP 文件来压缩文件。在更改文件时，FileSystemWatcher 用于获取信息。还解释了如何通过共享内存、命名管道和匿名管道进行通信。最后，讨论了如何把.NET 流映射到 Windows Runtime 流，在 Windows 应用程序中利用.NET 特性。

本书的其他章节会介绍流的操作。第 23 章在网络上使用流发送数据。读写 XML 文件和发送大型 XML 文件的内容参见网上附加第 2 章。

第 23 章将介绍联网和在网络上发送流的内容。

# 第23章 网络

**本章要点**

- 使用 HttpClient
- 操作 IP 地址，执行 DNS 查询
- 用 WebListener 创建服务器
- 用 TCP、UDP 和套接字类进行套接字编程

**本章源代码下载地址(wrox.com)：**

打开 www.wrox.com 的 Download Code 选项卡可下载本章源代码。源代码也可以在 Networking 目录的 https://github.com/ProfessionalCSharp/ProfessionalCSharp7 中找到。

本章代码分为以下几个主要的示例文件：

- HttpClientSample
- WinAppHttpClient
- HttpServer
- Utilities
- DnsLookup
- HttpClientUsingTcp
- TcpServer
- WinAppTcpClient
- UdpReceiver
- UdpSender
- SocketServer
- SocketClient

## 23.1 概述

本章将采取非常实用的网络方法，结合示例讨论相关理论和相应的网络概念。本章并不是计算机网络的指南，但会介绍如何使用.NET Framework 进行网络通信。

本章介绍了如何使用网络协议创建客户端和服务器。从最简单的示例开始，阐明如何给服务器发送请求和在响应中存储返回的信息。

然后讨论如何创建 HTTP 服务器，使用实用工具类分拆和创建 URI，把主机名解析为 IP 地址。还介绍了通过 TCP 和 UDP 收发数据，以及如何利用 Socket 类。

在网络环境下，我们最感兴趣的两个名称空间是 System.Net 和 System.Net.Sockets。System.Net 名称空间通常与较高层的操作有关，例如下载和上传文件，使用 HTTP 和其他协议进行 Web 请求等；而 System.Net.Sockets 名称空间包含的类通常与较低层的操作有关。如果要直接使用套接字或 TCP/IP 之类的协议，这个名称空间中的类就非常有用，这些类中的方法与 Windows 套接字(Winsock)API 函数(派生自 Berkeley 套接字接口)非常类似。本章介绍的一些对象位于 System.IO 名称空间中。

## 23.2 HttpClient 类

HttpClient 类用于发送 HTTP 请求，接收请求的响应。它在 System.Net.Http 名称空间中。System.Net.Http 名称空间中的类有助于简化在客户端和服务器上使用 Web 服务。

HttpClient 类派生于 HttpMessageInvoker 类，这个基类负责执行 SendAsync 方法。SendAsync 方法是 HttpClient 类的主干。如本节后面所述，这个方法有几个派生物。顾名思义，SendAsync 方法调用是异步的，这样就可以编写一个完全异步的系统来调用 Web 服务。

**警告：**

HttpClient 类实现了 IDisposable 接口。一般来说，实现 IDisposable 的对象应该在使用后销毁。HttpClient 类也是如此。但是，HttpClient 的 Dispose 方法不会立即释放相关的套接字，而是在超时后释放。这个超时可能需要 20 秒。有了这个超时，使用许多 HttpClient 对象实例可能导致程序耗尽套接字。解决方案是：构建 HttpClient 类，以进行重用。可以对许多请求使用这个类，而不是每次都创建一个新实例。

### 23.2.1 发出异步的 Get 请求

本章的下载代码示例是 HttpClientSample。它以不同的方式异步调用 Web 服务。为了演示本例使用的不同方法，使用了命令行参数。

示例代码使用了以下名称空间：

System
System.Linq
System.Net
System.Net.Http
System.Net.Http.Headers
System.Threading
System.Threading.Tasks

第一段代码实例化一个 HttpClient 对象，把它赋予私有字段_httpClient，以进行重用。这个 HttpClient 对象是线程安全的，所以一个 HttpClient 对象就可以用于处理多个请求。HttpClient 的每个实例都维护它自己的线程池，所以 HttpClient 实例之间的请求会被隔离。

接着调用 GetAsync，给它传递要调用的方法的地址，把一个 HTTP GET 请求发送给服务器。GetAsync 调用被重载为带一个字符串或 URI 对象。在本例中调用 Microsoft 的 OData 示例站点 http://services.odata.org，但可以修改这个地址，以调用任意多个 REST Web 服务。

对 GetAsync 的调用返回一个 HttpResponseMessage 对象。HttpResponseMessage 类表示包含标题、状态和内容的响应。检查响应的 IsSuccessfulStatusCode 属性，可以确定请求是否成功。如果调用成功，就使用 ReadAsStringAsync 方法把返回的内容检索为一个字符串(代码文件 HttpClientSample/HttpClientSamples.cs)：

```
private const string NorthwindUrl =
  "http://services.data.org/Northwind/Northwind.svc/Regions";
private const string IncorrectUrl =
  "http://services.data.org/Northwind1/Northwind.svc/Regions";

private HttpClient _httpClient;
public HttpClient HttpClient =>
  _httpClient ?? (_httpClient = new HttpClient());

private async Task GetDataSimpleAsync()
{
  HttpResponseMessage response = await HttpClient.GetAsync(NorthwindUrl);
  if(response.IsSuccessStatusCode)
    {
     Console.WriteLine($"Response Status Code: {(int)response.StatusCode} " +
       $"{response.ReasonPhrase}");
     string responseBodyAsText = await response.Content.ReadAsStringAsync();
     Console.WriteLine($"Received payload of {responseBodyAsText.Length} characters");
     Console.WriteLine();
     Console.WriteLine(responseBodyAsText);
    }
  }
}
```

用命令行参数-s 执行这段代码，产生以下输出：

```
Response Status Code: 200 OK
Received payload of 3379 characters
<?xml version="1.0" encoding="utf-8"?>
<!- ... ->
```

**注意：**

因为 HttpClient 类使用 GetAsync 方法调用，且使用了 await 关键字，所以返回调用线程，并可以执行其他工作。GetAsync 方法的结果可用时，就用该方法继续线程，响应写入 response 变量。await 关键字参见第 15 章，任务的创建和使用参见第 21 章。

## 23.2.2 抛出异常

如果调用 HttpClient 类的 GetAsync 方法失败，默认情况下不产生异常。调用 EnsureSuccessStatusCode 方法和 HttpResponseMessage，很容易改变这一点。该方法检查 IsSuccessStatusCode 是否是 false，否则就抛出一个异常(代码文件 HttpClientSample/HttpClientSamples.cs)：

```
private async Task GetDataWithExceptionsAsync()
{
  try
  {
    HttpResponseMessage response = await HttpClient.GetAsync(IncorrectUrl);
    response.EnsureSuccessStatusCode();
    ConsoleWriteLine($"Response Status Code: {(int)response.StatusCode} " +
      $"{response.ReasonPhrase}");
    string responseBodyAsText = await response.Content.ReadAsStringAsync();
    Console.WriteLine($"Received payload of {responseBodyAsText.Length} characters");
    Console.WriteLine();
    Console.WriteLine(responseBodyAsText);
  }
  catch (Exception ex)
  {
    Console.WriteLine($"{ex.Message}");
  }
}
```

## 23.2.3 传递标题

发出请求时没有设置或改变任何标题，但 HttpClient 的 DefaultRequestHeaders 属性允许设置或改变标题。使用 Add 方法可以给集合添加标题。设置标题值后，标题和标题值会与这个 HttpClient 实例发送的每个请求一起发送。

响应内容默认为 XML 格式。要改变它，可以在请求中添加一个 Accept 标题，以使用 JSON。在调用 GetAsync 之前添加如下代码，内容就会以 JSON 格式返回：

```
client.DefaultRequestHeaders.Add("Accept", "application/json;odata=verbose");
```

添加和删除标题，运行示例，会以 XML 和 JSON 格式显示内容。

从 DefaultHeaders 属性返回的 HttpRequestHeaders 对象有许多辅助属性，可用于许多标准标题。可以从这些属性中读取标题的值，但它们是只读的。要设置其值，需要使用 Add 方法。在代码片段中，添加了 HTTP Accept 标题。根据服务器接收到的 Accept 标题，服务器可以基于客户的需求返回不同的数据格式。发送 Accept 标题 application/json 时，客户就通知服务器，它接受 JSON 格式的数据。标题信息用 ShowHeaders 方法显示，从服务器接收响应时，也调用该方法(代码文件 HttpClientSample/HttpClientSamples.cs)：

```
public Task GetDataWithHeadersAsync()
{
  try
  {
    HttpClient.DefaultRequestHeaders.Add("Accept",
      "application/json;odata=verbose");
    ShowHeaders("Request Headers:", HttpClient.DefaultRequestHeaders);

    HttpResponseMessage response = await client.GetAsync(NorthwindUrl);
    HttpClient.EnsureSuccessStatusCode();
    ShowHeaders("Response Headers:", response.Headers);
    //...
  }
}
```

与上一个示例不同，添加了 ShowHeaders 方法，它把一个 HttpHeaders 对象作为参数。HttpHeaders 是 HttpRequestHeaders 和 HttpResponseHeaders 的基类。这两个特殊化的类都添加了辅助属性，以直接访问标题。HttpHeader 对象定义为 keyValuePair<string, IEnumerable<string>>。这表示每个标题在集合中都可以有多个值。因此，如果希望改变标题中的值，就需要删除原值，添加新值。

ShowHeaders 函数很简单，它迭代 HttpHeaders 中的所有标题。枚举返回 KeyValuePair<string, IEnumerable<string>>元素，为每个键显示值的字符串版本：

```
public void ShowHeaders(string title, HttpHeaders headers)
{
  Console.WriteLine(title);
  foreach (var header in headers)
  {
    string value = string.Join(" ", header.Value);
    Console.WriteLine($"Header: {header.Key} Value: {value}");
  }
  Console.WriteLine();
}
```

运行这段代码，就显示请求的任何标题。

```
Request Headers:
Header: Accept Value: application/json; odata=verbose
Response Headers:
Header: Cache-Control Value: private
Header: Date Value: Thu, 31 Aug 2017 09:58:09 GMT
Header: Server Value: Microsoft-IIS/8.0
Header: Set-Cookie Value:
ARRAffinity=a5ee7717b148daedb0164e6e19088a5a78c47693a6
0e57422887d7e011fb1e5e;Path=/;Domain=services.odata.org
Header: Vary Value: *
Header: X-Content-Type-Options Value: nosniff
Header: DataServiceVersion Value: 2.0;
Header: X-AspNet-Version Value: 4.0.30319
Header: X-Powered-By Value: ASP.NET
Header: Access-Control-Allow-Origin Value: *
Header: Access-Control-Allow-Methods Value: GET
Header: Access-Control-Allow-Headers Value: Accept, Origin, Content-Type,
MaxDataServiceVersion
Header: Access-Control-Expose-Headers Value: DataServiceVersion
```

因为现在客户端请求 JSON 数据，服务器返回 JSON，也可以看到这些信息：

```
Response Status Code: 200 OK
```

```
Received payload of 1551 characters
{"d":{"results":[{"__metadata":{"id":"http://services.odata.org/Northwind/
Northwind.svc/Regions(1) ", "uri":
```

### 23.2.4　访问内容

先前的代码片段展示了如何访问 Content 属性，获取一个字符串。响应中的 Content 属性返回一个 HttpContent 对象。为了获得 HttpContent 对象中的数据，需要使用所提供的一个方法。在例子中，使用了 ReadAsStringAsync 方法。它返回内容的字符串表示。顾名思义，这是一个异步调用。除了使用 async 关键字之外，也可以使用 Result 属性。调用 Result 属性会阻塞该调用，直到 ReadAsStringAsync 方法执行完毕，然后继续执行下面的代码。

其他从 HttpContent 对象中获得数据的方法有 ReadAsByteArrayAsync(返回数据的字节数组)和 ReadAsStreamAsync(返回一个流)。也可以使用 LoadIntoBufferAsync 把内容加载到内存缓存中。

Headers 属性返回 HttpContentHeaders 对象。它的工作方式与前面例子中的请求和响应标题相同。

**注意：**

除了使用 HttpClient 和 HttpContent 类的 GetAsync 和 ReadAsStringAsync 方法之外，HttpClient 类还提供了方法 GetStringAsync，来返回一个字符串，而不需要调用两个方法。然而使用这个方法时，对错误状态和其他信息没有那么多的控制。

**注意：**

流参见第 22 章。

### 23.2.5　用 HttpMessageHandler 自定义请求

HttpClient 类可以把 HttpMessageHandler 作为其构造函数的参数，这样就可以定制请求。可以传递 HttpClientHandler 的实例。它有许多属性可以设置，例如 ClientCertificates、Pipelining、CachePolity、ImpersonationLevel 等。

下一个代码片段实例化 SampleMessageHandler，并传递给 HttpClient 构造函数(代码文件 HttpClientSample/HttpClientSamples.cs)：

```
private HttpClient _httpClientWithMessageHandler;
public HttpClient HttpClientWithMessageHandler =>
  _httpClientWithMessageHandler ?? (_httpClientWithMessageHandler =
    new HttpClient(new SampleMessageHandler("error")));
```

这个处理程序类型 SampleMessageHandler 的作用是把一个字符串作为参数，在控制台上显示它，如果消息是“error”，就把响应的状态码设置为 Bad Request。如果创建一个派生于 HttpClientHandler 的类，就可以重写一些属性和方法 SendAsync。SendAsync 通常会重写，因为发送到服务器的请求会受影响。如果_displayMessage 设置为“error”，就返回一个 HttpResponseMessage 和错误请求。该方法需要返回一个 Task。对于错误的情况，不需要调用异步方法；这就是为什么只是用 Task.FromResult 返回错误 (代码文件 HttpClientSample/SampleMessageHandler.cs)：

```
public class SampleMessageHandler : HttpClientHandler
{
  private string _message;
  public SampleMessageHandler(string message) =>
    _message = message;

  protected override Task<HttpResponseMessage> SendAsync(
    HttpRequestMessage request, CancellationToken cancellationToken)
  {
    Console.WriteLine($"In SampleMessageHandler {_message}");
    if(_message == "error")
    {
      var response = new HttpResponseMessage(HttpStatusCode.BadRequest);
      return Task.FromResult<HttpResponseMessage>(response);
```

```
      }
    return base.SendAsync(request, cancellationToken);
    }
  }
```

添加定制处理程序有许多理由。设置处理程序管道，是为了添加多个处理程序。除了默认的处理程序之外，还有 DelegatingHandler，它执行一些代码，再把调用委托给内部的处理程序或下一个处理程序。HttpClientHandler 是最后一个处理程序，它把请求发送到地址。图 23-1 显示了管道。每个添加的 DelegatingHandler 都调用下一个或内部的处理程序，最后一个是基于 HttpClientHandler 的处理程序。

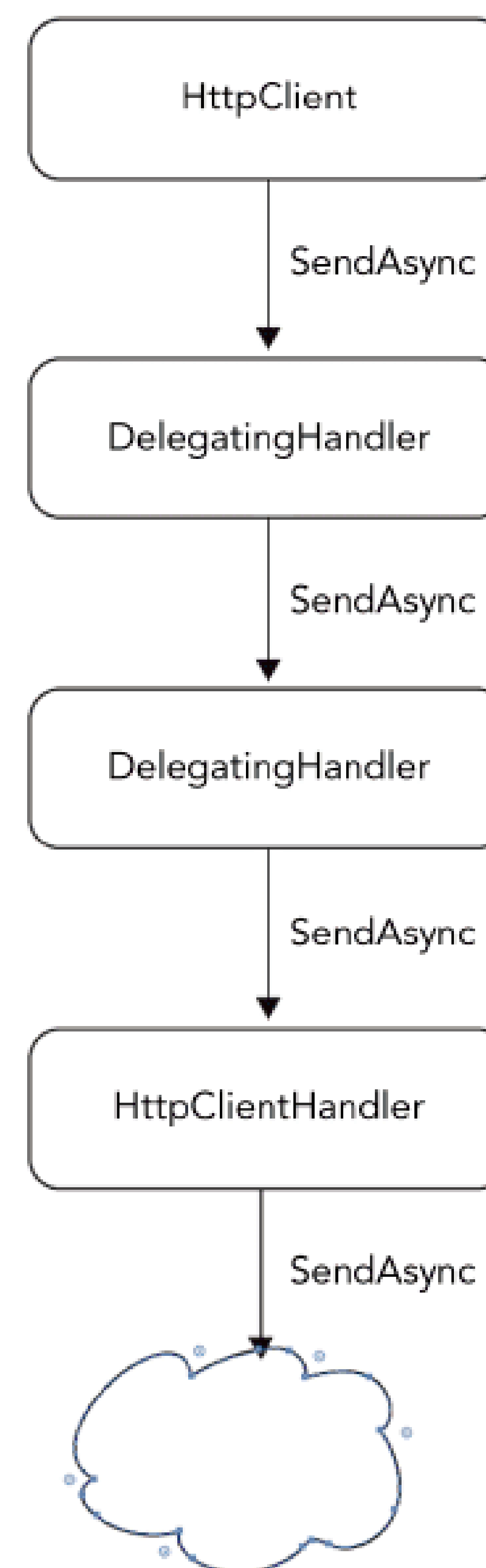


图 23-1

### 23.2.6 使用 SendAsync 创建 HttpRequestMessage

在后台，HttpClient 类的 GetAsync 方法调用 SendAsync 方法。除了使用 GetAsync 方法之外，还可以使用 SendAsync 方法发送一个 HTTP 请求。使用 SendAsync，可以对定义请求有更多的控制。重载 HttpRequestMessage 类的构造函数，传递 HttpMethod 的一个值。GetAsync 方法用 HttpMethod.Get 创建一个 HTTP 请求。使用 HttpMethod，不仅可以发送 GET、POST、PUT 和 DELETE 请求，也可以发送 HEAD、OPTIONS 和 TRACE。有了 HttpRequestMessage 对象，可以用 HttpClient 调用 SendAsync 方法(代码文件 HttpClientSample/HttpClientSamples.cs)：

```
private async Task GetDataAdvancedAsync()
{
  var request = new HttpRequestMessage(HttpMethod.Get, NorthwindUrl);
  HttpResponseMessage response = await client.SendAsync(request);
  //...
}
```

**注意：**
本章只使用 HttpClient 类发出 HTTP GET 请求。HttpClient 类还允许使用 PostAsync、PutAsync 和 DeleteAsync 方法，发送 HTTP POST、PUT 和 DELETE 请求。这些方法在第 32 章使用，发出这些请求，在 Web 服务中调用相应的动作方法。

创建 HttpRequestMessage 对象后，可以使用 Headers 和 Content 属性提供标题和内容。使用 Version 属性，可以指定 HTTP 版本。

**注意：**
HTTP/1.0 在 1996 年发布，几年后发布了 1.1 版本。在 1.0 版本中，服务器返回数据后，连接总是关闭；在 1.1 版本中，增加了 keep-alive 标题，允许客户端根据需要保持连接打开，因为客户端可能希望发出更多的请求，不仅接收 HTML 代码，还接收 CSS、JavaScript 文件和图片。1999 年定义了 HTTP/1.1 后，过了 16 年，HTTP/2 才在 2015 年完成。版本 2 有什么优点？HTTP/2 允许在相同的连接上发出多个并发请求，压缩标题信息，客户机可以定义哪个资源更重要，服务器可以通过服务器推操作把资源发送到客户端。HTTP/2 支持服务器推送，意味着一旦 HTTP/2 支持无处不在，WebSockets 就会过时。所有浏览器的新版本，以及运行在 Windows 10 和 Windows Server 2016 上的 IIS，都支持 HTTP/2。对于.NET Core，(在撰写本文时)HTTP/2 计划成为未来的里程碑；参见 https://github.com/dotnet/corefx/issues/23134。

### 23.2.7 使用 HttpClient 和 Windows Runtime

在撰写本书时，用于控制台应用程序和 WPF 的 HttpClient 类不支持 HTTP/2。然而，用于通用 Windows 平台的 HttpClient 类有不同的实现，它基于 Windows 10 API 的功能。因此，HttpClient 支持 HTTP/2，甚至在默认情况下使用这个版本。

下一个代码示例显示了一个通用 Windows 应用程序，它向进入一个文本框的链接发出一个 HTTP 请求，并

显示结果，给出 HTTP 版本信息。以下代码片段显示了 XAML 代码，图 23-2 显示了应用程序的用户界面(代码文件 WinAppHttpClient/MainPage.xaml)：

```
<StackPanel Orientation="Horizontal">
  <TextBox Header="Url" Text="{x:Bind Url, Mode=TwoWay}" MinWidth="200"
    Margin="5" />
  <Button Content="Send" Click="{x:Bind OnSendRequest}" Margin="10,5,5,5"
    VerticalAlignment="Bottom" />
</StackPanel>
<TextBox Header="Version" Text="{x:Bind Version, Mode=OneWay}" Grid.Row="1"
  Margin="5" IsReadOnly="True" />
<TextBox AcceptsReturn="True" IsReadOnly="True"
  Text="{x:Bind Result, Mode=OneWay}" Grid.Row="2"
  ScrollViewer.HorizontalScrollBarVisibility="Auto"
  ScrollViewer.VerticalScrollBarVisibility="Auto" />
```

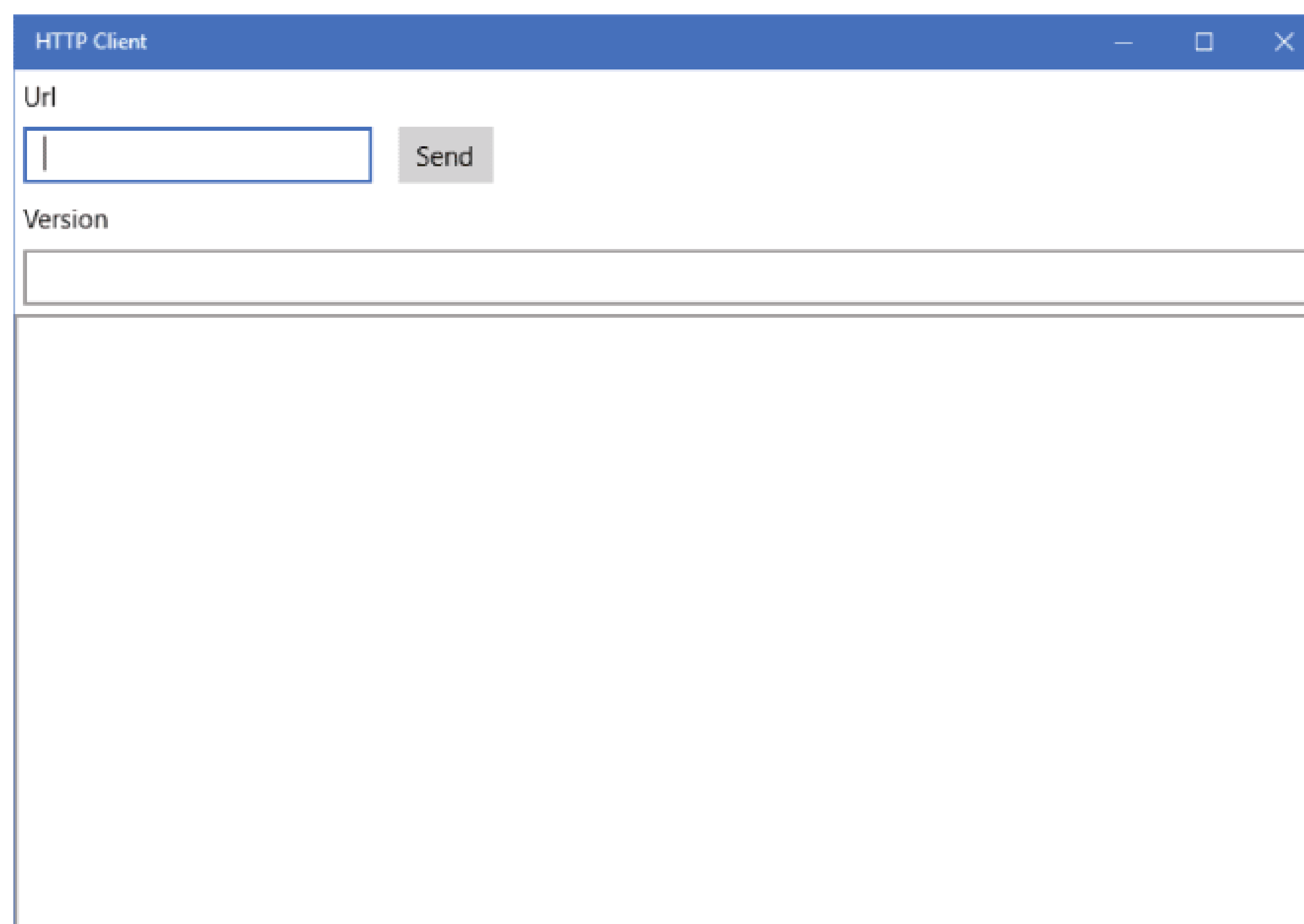



图 23-2

**注意：**
XAML 代码和依赖属性参见第 33 章，编译后的绑定参见第 34 章。

属性 Url、Version 和 Result 实现为依赖属性，以自动更新 UI。下面的代码片段显示了 Url 属性(代码文件 WinAppHttpClient/MainPage.xaml.cs)：

```
public string Url
{
  get => (string)GetValue(UrlProperty);
  set => SetValue(UrlProperty, value);
}

public static readonly DependencyProperty UrlProperty =
  DependencyProperty.Register("Url", typeof(string), typeof(MainPage),
    new PropertyMetadata(string.Empty));
```

HttpClient 类用于 OnSendRequest 方法。单击 UI 中的 Send 按钮，就调用该方法。在前面的示例中，SendAsync 方法用于发出 HTTP 请求。为了看到请求确实是使用 HTTP/2 版本发出的，可以在调试器中检查 request.Version 属性。服务器给出的版本是来自 response. Version，并写入在 UI 中绑定的 Version 属性。如今，大多数服务器都只支持 HTTP 1.1 版本。如前所述，Windows Server 2016 支持 HTTP/2：

```
private async void OnSendRequest()
{
  try
  {
    using (var client = new HttpClient())
    {
      var request = new HttpRequestMessage(HttpMethod.Get, Url);
      HttpResponseMessage response = await client.SendAsync(request);
```

```
        Version = response.Version.ToString();
        response.EnsureSuccessStatusCode();
        Result = await response.Content.ReadAsStringAsync();
      }
    }
    catch (Exception ex)
    {
      await new MessageDialog(ex.Message).ShowAsync();
    }
  }
```

运行该应用程序，向 https://http2.akamai.com/demo 发出请求，就返回 HTTP/2。

## 23.3 使用 WebListener 类

使用 IIS(Internet Information Service，互联网信息服务)作为 HTTP 服务器通常是一个好方法，因为可以访问很多功能，如可伸缩性、健康监测、用于管理的图形用户界面等。然而，也可以轻松创建自己的简单 HTTP 服务器。自.NET Framework 2.0 以来，就可以使用 HttpListener，但是现在自从.NET Core 1.0 以来，有一个新的 WebListener 类。

HttpServer 的示例代码使用了以下依赖项和名称空间：

**依赖项**

Microsoft.Net.Http.Server

**名称空间**

Microsoft.Net.Http.Server

System

System.Collections.Generic

System.Linq

System.Net

System.Reflection

System.Text

System.Threading.Tasks

HTTP 服务器的示例代码是一个控制台应用程序(包)，允许传递一个 URL 前缀的列表，来定义服务器侦听的地点。这类前缀的一个例子是 http://localhost:8082/samples，其中如果路径以 samples 开头，服务器就只侦听本地主机上的端口 8082。不管其后的路径是什么，服务器都处理请求。不仅支持来自本地主机的请求，还可以使用+字符，比如 http://+:8082/samples。这样，服务器也可以从所有的主机名中访问。如果不以提升模式启动 Visual Studio，运行侦听器的用户就需要许可。为此，可以以提升模式运行一个命令提示符，使用如下 netsh 命令来添加 URL：

```
>netsh http add urlacl url=http://+:8082/samples user=Everyone
```

示例代码检查参数是否传递了至少一个前缀，之后调用 StartServer 方法(代码文件 HttpServer/Program.cs):

```
static async Task Main(string[] args)
{
  if (args.Length < 1)
  {
    ShowUsage();
    return;
  }
  await StartServerAsync(args);
  Console.ReadLine();
}

private static void ShowUsage()
{
  Console.WriteLine("Usage: HttpServer Prefix [Prefix2] [Prefix3] [Prefix4]");
}
```

该程序的核心是 StartServer 方法。这里实例化 WebListener 类，添加在命令参数列表中定义的前缀。调用 WebListener 类的 Start 方法，注册系统上的端口。接下来，调用 GetContextAsync 方法后，侦听器等待客户端连接和发送数据。一旦客户端发送了 HTTP 请求，请求就可以读取 GetContextAsync 返回 的 HttpContext 对象。对于来自客户端的请求和发回的回应，都使用 HttpContext 对象。Request 属性返回一个 Request 对象。Request 对象包含 HTTP 标题信息。在 HTTP POST 请求中，Request 还包含请求体。Response 属性返回 Response 对象，它允许返回标题信息(使用 Headers 属性)、状态码(StatusCode 属性)和响应体(Body 属性)：

```
public static async Task StartServerAsync(params string[] prefixes)
{
  try
  {
    Console.WriteLine($"server starting at");
    var listener = new WebListener();
    foreach (var prefix in prefixes)
    {
      listener.UrlPrefixes.Add(prefix);
      Console.WriteLine($"\t{prefix}");
    }
    listener.Start();
    do
    {
      using (RequestContext context = await listener.GetContextAsync())
      {
        context.Response.Headers.Add("content-type",
        new string[] { "text/html" });
        context.Response.StatusCode = (int)HttpStatusCode.OK;
        byte[] buffer = GetHtmlContent(context.Request);
        await context.Response.Body.WriteAsync(buffer, 0, buffer.Length);
      }
    } while (true);
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

示例代码返回一个 HTML 文件，使用 GetHtmlContent 方法检索它。这个方法利用 htmlFormat 格式字符串，该字符串在标题和正文中有两个占位符。GetHtmlContent 方法使用 string.Format 方法填充占位符。为了填充 HTML 体，使用两个辅助方法 GetHeaderInfo 和 GetRequestInfo，检索请求的标题信息和 Request 对象的所有属性值：

```
private static string htmlFormat =
  "<!DOCTYPE html><html><head><title>{0}</title></head>" +
  "<body>{1}</body></html>";

private static byte[] GetHtmlContent(Request request)
{
  string title = "Sample WebListener";
  var sb = new StringBuilder("<h1>Hello from the server</h1>");
  sb.Append("<h2>Header Info</h2>");
  sb.Append(string.Join(" ", GetHeaderInfo(request.Headers)));
  sb.Append("<h2>Request Object Information</h2>");
  sb.Append(string.Join(" ", GetRequestInfo(request)));
  string html = string.Format(htmlFormat, title, sb.ToString());
  return Encoding.UTF8.GetBytes(html);
}
```

GetHeaderInfo 方法从 HeaderCollection 中检索键和值，返回一个 div 元素，其中包含了每个键和值：

```
private static IEnumerable<string> GetHeaderInfo(HeaderCollection headers) =>
  headers.Keys.Select(key =>
    $"<div>{key}: {string.Join(",", headers.GetValues(key))}</div>");
```

GetRequestInfo 方法利用反射获得 Request 类型的所有属性，返回属性名称及其值：

```
private static IEnumerable<string> GetRequestInfo(Request request) =>
  request.GetType().GetProperties().Select(
    p => $"<div>{p.Name}: {p.GetValue(request)}</div>");
```

**注意：**

GetHeaderInfo 和 GetRequestInfo 方法利用表达式体的成员函数、LINQ 和反射。表达式体的成员函数参见第 3 章。第 12 章讨论了 LINQ。第 16 章把反射作为一个重要的话题。

运行服务器，使用 Google Chrome 等浏览器，通过 URL 访问服务器，如 http://[hostname]:8082/samples/Hello?sample=text，结果输出如图 23-3 所示。

**注意：**

要从 Edge 浏览器访问 localhost，需要启用 localhost 回溯，这可以通过 about:flags 实现。在某些 Windows 10 构建版本中，如果启用了 localhost 回溯，就会出现从 Edge 浏览器事件中访问 localhost 的问题。在这种情况下，应使用其他浏览器，如 Internet Explorer。

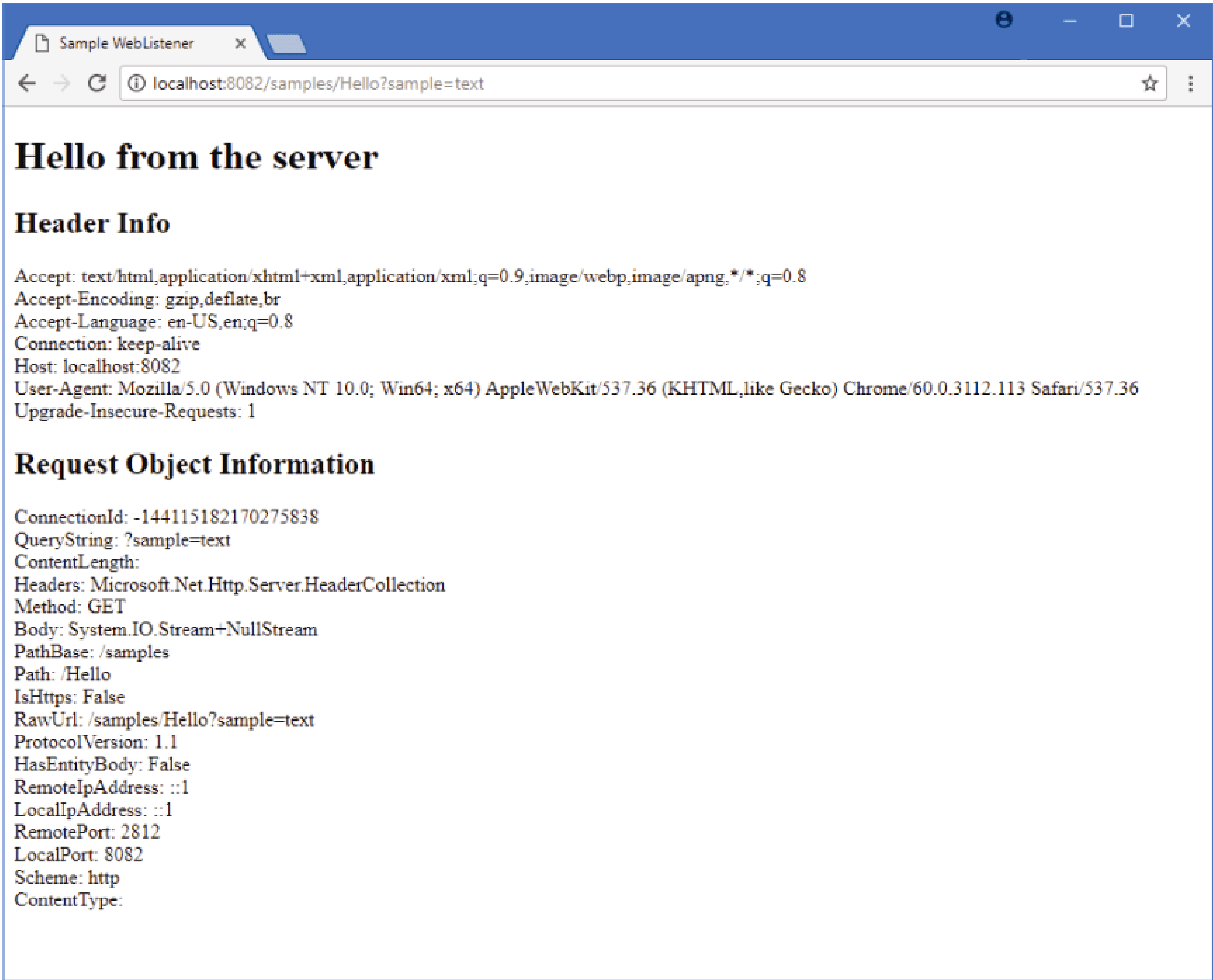



图 23-3

# 23.4 使用实用工具类

在使用抽象 HTTP 协议的类，如 HttpClient 和 WebListener，处理 HTTP 请求和响应后，下面看看一些实用工具类，它们在处理 URI 和 IP 地址时，更容易进行 Web 编程。

在Internet上，服务器和客户端都由IP地址或主机名(也称作DNS名称)标识。通常，主机名是在Web浏览器的窗口中输入的友好名称，如www.wrox.com或www.cninnovation.com等。另一方面，IP地址是计算机用于互相识别的标识符，它实际上是用于确保Web请求和响应到达相应计算机的地址。一台计算机甚至可以有多个IP地址。

目前，IP 地址一般是一个 32 位或 128 位的值，这取决于使用的是 IPv4 还是 IPv6。例如 192.168.1.100 就是一个 32 位的 IP 地址。目前有许多计算机和其他设备在竞争 Internet 上的一个地点，所以人们开发了 IPv6。IPv6 至多可以提供 $3\times10^{28}$ 个不同的地址。.NET 允许应用程序同时使用 IPv4 和 IPv6。

为了使这些主机名发挥作用，首先必须发送一个网络请求，把主机名翻译成 IP 地址，翻译工作由一个或几

个 DNS 服务器完成。DNS 服务器中保存的一个表把主机名映射为它知道的所有计算机的 IP 地址以及其他 DNS 服务器的 IP 地址，这些 DNS 服务器用于在该表中查找它不知道的主机名。本地计算机至少要知道一个 DNS 服务器。网络管理员在设置计算机时配置该信息。

在发送请求之前，计算机首先应要求 DNS 服务器指出与输入的主机名相对应的 IP 地址。找到正确的 IP 地址后，计算机就可以定位请求，并通过网络发送它。所有这些工作一般都在用户浏览 Web 时在后台进行。

.NET Framework 提供了许多能够帮助寻找 IP 地址和主机信息的类。

示例代码使用了以下名称空间：

System

System.Net

## 23.4.1　URI

Uri 和 UriBuilder 是 System 名称空间中的两个类，它们都用于表示 URI。Uri 类允许分析、组合和比较 URI。而 UriBuilder 类允许把给定的字符串当作 URI 的组成部分，从而构建一个 URI。

下面的代码片段演示了 Uri 类的特性。构造函数可以传递相对和绝对 URL。这个类定义了几个只读属性，来访问 URL 的各个部分，例如模式、主机名、端口号、查询字符串和 URL 的各个部分(代码文件 Utilities/Program.cs)：

```
public static void UriSample(string url)
{
  var page = new Uri(url);
  Console.WriteLine($"scheme: {page.Scheme}");
  Console.WriteLine($"host: {page.Host}, type: {page.HostNameType}, " +
    $"idn host: {page.IdnHost}");
  Console.WriteLine($"port: {page.Port}");
  Console.WriteLine($"path: {page.AbsolutePath}");
  Console.WriteLine($"query: {page.Query}");

  foreach (var segment in page.Segments)
  {
    Console.WriteLine($"segment: {segment}");
  }
  //...
}
```

运行应用程序，传递下面的 URL 和包含一个路径和查询字符串的字符串：http://www.amazon.com/Professional-C-6-0-Christian-Nagel/dp/111909660X/ref=sr_1_4?ie=UTF8&qid=1438459506&sr=8-4&keywords=professional+c%23+6。

将得到下面的输出：

```
scheme: http
host: www.amazon.com, type: Dns, idn host: www.amazon.com
port: 80
path: /Professional-C-6-0-Christian-Nagel/dp/111909660X/ref=sr_1_4
query: ?ie=UTF8&qid=1438459506&sr=8-4&keywords=professional+c%23+6
segment: /
segment: Professional-C-6-0-Christian-Nagel/
segment: dp/
segment: 111909660X/
segment: ref=sr_1_4
```

与 Uri 类不同，UriBuilder 定义了读-写属性，如下面的代码片段所示。可以创建一个 UriBuilder 实例，指定这些属性，并得到一个从 Uri 属性返回的 URL：

```
public static void UriSample(string url)
{
  // ...
  var builder = new UriBuilder();
  builder.Host = "www.cninnovation.com";
  builder.Port = 80;
  builder.Path = "training/MVC";
  Uri uri = builder.Uri;
  Console.WriteLine(uri);
}
```

除了使用 UriBuilder 的属性之外，这个类还提供了构造函数的几个重载版本，其中也可以传递 URL 的各个部分。

### 23.4.2 IPAddress

IPAddress 类代表 IP 地址。使用 GetAddressBytes 属性可以把地址本身作为字节数组，并使用 ToString()方法转换为用小数点隔开的十进制格式。此外，IPAddress 类也实现静态的 Parse()和 TryParse 方法，这两个方法的作用与 ToString()方法正好相反，把小数点隔开的十进制字符串转换为 IPAddress。代码示例也访问 AddressFamily 属性，并将一个 IPv4 地址转换成 IPv6，反之亦然(代码文件 Utilities/Program.cs)：

```
public static void IPAddressSample(string ipAddressString)
{
  IPAddress address;
  if (!IPAddress.TryParse(ipAddressString, out address))
  {
    Console.WriteLine($"cannot parse {ipAddressString}");
    return;
  }
  byte[] bytes = address.GetAddressBytes();
  for (int i = 0; i < bytes.Length; i++)
  {
    Console.WriteLine($"byte {i}: {bytes[i]:X}");
  }
  Console.WriteLine($"family: {address.AddressFamily}, " +
    $"map to ipv6: {address.MapToIPv6()}, map to ipv4: {address.MapToIPv4()}");
  // ...
```

给方法传递地址 65.52.128.33，输出结果如下：

```
byte 0: 41
byte 1: 34
byte 2: 80
byte 3: 21
family: InterNetwork, map to ipv6: ::ffff:65.52.128.33, map to ipv4: 65.52.128.3
3
```

IPAddress 类也定义了静态属性，来创建特殊的地址，如 loopback、broadcast 和 anycast：

```
public static void IPAddressSample(string ipAddressString)
{
  //...
  Console.WriteLine($"IPv4 loopback address: {IPAddress.Loopback}");
  Console.WriteLine($"IPv6 loopback address: {IPAddress.IPv6Loopback}");
  Console.WriteLine($"IPv4 broadcast address: {IPAddress.Broadcast}");
  Console.WriteLine($"IPv4 any address: {IPAddress.Any}");
  Console.WriteLine($"IPv6 any address: {IPAddress.IPv6Any}");
}
```

通过 loopback 地址，可以绕过网络硬件。这个 IP 地址代表主机名 localhost。

每个 broadcast 地址都在本地网络中寻址每个节点。这类地址不能用于 IPv6，因为这个概念不用于互联网协议的更新版本。最初定义 IPv4 后，给 IPv6 添加了多播。通过多播，寻址一组节点，而不是所有节点。在 IPv6 中，多播完全取代广播。本章后面使用 UDP 时，会在代码示例中演示广播和多播。

通过 anycast，也使用一对多路由，但数据流只传送到网络上最近的节点。这对负载平衡很有用。对于 IPv4，Border Gateway Protocol (BGP)路由协议用来发现网络中的最短路径；对于 IPv6，这个功能是内置的。

运行应用程序时，可以看到下面的 IPv4 和 IPv6 地址：

```
IPv4 loopback address: 127.0.0.1
IPv6 loopback address: ::1
IPv4 broadcast address: 255.255.255.255
IPv4 any address: 0.0.0.0
IPv6 any address: ::
```

### 23.4.3 IPHostEntry

IPHostEntry 类封装与某台特定的主机相关的信息。通过这个类的 HostName 属性(这个属性返回一个字符串)，可以使用主机名；通过 AddressList 属性返回一个 IPAddress 对象数组。下一个示例使用 IPHostEntry 类。

### 23.4.4 Dns

Dns 类能够与默认的 DNS 服务器进行通信，以检索 IP 地址。

DnsLookup 示例代码使用了以下名称空间：

System

System.Net

System.Threading.Tasks

样例应用程序实现为一个控制台应用程序(包)，要求用户输入主机名(也可以添加一个 IP 地址)，通过 Dns.GetHostEntryAsync 得到一个 IPHostEntry。在 IPHostEntry 中，使用 AddressList 属性访问地址列表。主机的所有地址以及 AddressFamily 都写入控制台(代码文件 DnsLookup/Program.cs)：

```
static void Main()
{
  do
  {
    Console.Write("Hostname:\t");
    string hostname = ReadLine();
    if (hostname.CompareTo("exit") == 0)
    {
      Console.WriteLine("bye!");
      return;
    }

    OnLookupAsync(hostname).Wait();
    Console.WriteLine();
  } while (true);
}

public static async Task OnLookupAsync(string hostname)
{
  try
  {
    IPHostEntry ipHost = await Dns.GetHostEntryAsync(hostname);
    Console.WriteLine($"Hostname: {ipHost.HostName}");
    foreach (IPAddress address in ipHost.AddressList)
    {
      Console.WriteLine($"Address Family: {address.AddressFamily}");
      Console.WriteLine($"Address: {address}");
    }
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

运行应用程序，并输入几个主机名，得到如下输出。对于主机名 www.orf.at，可以看到这个主机名定义了多个 IP 地址。

```
Hostname: www.cninnovation.com
Hostname: www.cninnovation.com
Address Family: InterNetwork
Address: 65.52.128.33
Hostname: www.orf.at
Hostname: www.orf.at
Address Family: InterNetwork
Address: 194.232.104.150
Address Family: InterNetwork
Address: 194.232.104.139
Address Family: InterNetwork
Address: 194.232.104.140
Address Family: InterNetwork
Address: 194.232.104.142
Address Family: InterNetwork
Address: 194.232.104.141
Address Family: InterNetwork
Address: 194.232.104.149
Hostname: exit
bye!
```

**注意：**
Dns 类是比较有限的，例如不能指定使用非默认的 DNS 服务器。此外，IPHostEntry 的 Aliases 属性不在 GetHostEntryAsync 方法中填充。它只在 Dns 类的过时方法中填充，而且这些方法也不完全地填充这个属性。要充分利用 DNS 查找功能，最好使用第三方库。

下面介绍一些低级协议，如 TCP 和 UDP 等。

# 23.5 使用 TCP

HTTP 协议基于传输控制协议(Transmission Control Protocol，TCP)。要使用 TCP，客户端首先需要打开一个到服务器的连接，才能发送命令。而使用 HTTP，发送文本命令。HttpClient 和 WebListener 类隐藏了 HTTP 协议的细节。使用 TCP 类发送 HTTP 请求时，需要更多地了解 HTTP 协议。TCP 类没有提供用于 HTTP 协议的功能，必须自己提供。另一方面，TCP 类提供了更多的灵活性，因为可以使用这些类与基于 TCP 的其他协议。

传输控制协议(TCP)类为连接和发送两个端点之间的数据提供了简单的方法。端点是 IP 地址和端口号的组合。已有的协议很好地定义了端口号，例如，HTTP 使用端口 80，而 SMTP 使用端口 23。Internet 地址编码分配机构(Internet Assigned Numbers Authority，IANA，http://www.iana.org/)把端口号赋予这些已知的服务。除非实现某个已知的服务，否则应选择大于 1024 的端口号。

TCP 流量构成了目前 Internet 上的主要流量。TCP 通常是首选的协议，因为它提供了有保证的传输、错误校正和缓冲。TcpClient 类封装了 TCP 连接，提供了许多属性来控制连接，包括缓冲、缓冲区的大小和超时。通过 GetStream()方法请求 NetworkStream 对象可以实现读写功能。

TcpListener 类用 Start()方法侦听引入的 TCP 连接。当连接请求到达时，可以使用 AcceptSocket()方法返回一个套接字，与远程计算机通信，或使用 AcceptTcpClient()方法通过高层的 TcpClient 对象进行通信。阐明 TcpListener 类和 TcpClient 类如何协同工作的最简单方式是给出一个示例。

## 23.5.1 使用 TCP 创建 HTTP 客户程序

首先，创建一个控制台应用程序(包)，向 Web 服务器发送一个 HTTP 请求。以前用 HttpClient 类实现了这个功能，但使用 TcpClient 类需要深入 HTTP 协议。

HttpClientUsingTcp 示例代码使用了以下名称空间：

System
System.IO
System.Net.Sockets
System.Text
System.Threading.Tasks

应用程序接受一个命令行参数，传递服务器的名称。这样，就调用 RequestHtmlAsync 方法，向服务器发出 HTTP 请求。它用 Task 的 Result 属性返回一个字符串 (代码文件 HttpClientUsingTcp/Program.cs)：

```
static void Main(string[] args)
{
  if (args.Length != 1)
  {
    ShowUsage();
  }
  Task<string> t1 = RequestHtmlAsync(args[0]);
  Console.WriteLine(t1.Result);
  Console.ReadLine();
}

private static void ShowUsage()
{
  Console.WriteLine("Usage: HttpClientUsingTcp hostname");
}
```

现在看看 RequestHtmlAsync 方法的最重要部分。首先，实例化一个 TcpClient 对象。其次，使用 ConnectAsync 方法，在 HTTP 默认端口 80 上建立到主机的 TCP 连接。再次，通过 GetStream 方法检索一个流，使用这个连接进行读写：

```
private const int ReadBufferSize = 1024;
public static async Task<string> RequestHtmlAsync(string hostname)
{
  try
  {
    using (var client = new TcpClient())
    {
      await client.ConnectAsync(hostname, 80);
      NetworkStream stream = client.GetStream();
      //...
    }
  }
}
```

流现在可以用来把请求写到服务器，读取响应。HTTP 是一种基于文本的协议，所以很容易在字符串中定义请求。为了向服务器发出一个简单的请求，标题定义了 HTTP 方法 GET，其后是 URL/的路径和 HTTP 版本 HTTP/1.1。第二行定义了 Host 标题、主机名和端口号，第三行定义了 Connection 标题。通常，通过 Connection 标题，客户端请求 keep-alive，要求服务器保持连接打开，因为客户端希望发出更多的请求。这里只向服务器发出一个请求，所以服务器应该关闭连接，从而 close 设置为 Connection 标题。为了结束标题信息，需要使用\r\n 给请求添加一个空行。标题信息调用 NetworkStream 的方法 WriteAsync，用 UTF-8 编码发送。\r\n 为了立即向服务器发送缓存，请调用 FlushAsync 方法。否则数据就可能保存在本地缓存：

```
//...
string header = "GET/HTTP/1.1\r\n" +
$"Host: {hostname}:80\r\n" +
"Connection: close\r\n" +
"\r\n";
byte[] buffer = Encoding.UTF8.GetBytes(header);
await stream.WriteAsync(buffer, 0, buffer.Length);
await stream.FlushAsync();
```

现在可以继续这个过程，从服务器中读取回应。不知道回应有多大，所以创建一个动态生长的 MemoryStream。使用 ReadAsync 方法把服务器的回应暂时写入一个字节数组，这个字节数组的内容添加到 MemoryStream 中。从服务器中读取所有数据后，StreamReader 接管控制，把数据从流读入一个字符串，并返回给调用者：

```
  var ms = new MemoryStream();
  buffer = new byte[ReadBufferSize];
  int read = 0;
  do
  {
    read = await stream.ReadAsync(buffer, 0, ReadBufferSize);
    ms.Write(buffer, 0, read);
    Array.Clear(buffer, 0, buffer.Length);
  } while (read > 0);
  ms.Seek(0, SeekOrigin.Begin);
  var reader = new StreamReader(ms);
  return reader.ReadToEnd();
}
//...
```

把一个网站传递给程序，会看到一个成功的请求，其 HTML 内容显示在控制台上。

现在该创建一个 TCP 侦听器和自定义协议了。

## 23.5.2　创建 TCP 侦听器

创建基于 TCP 的自定义协议需要对架构进行一些思考。可以定义自己的二进制协议，每个位都保存在数据传输中，但读取比较复杂，或者可以使用基于文本的格式，例如 HTTP 或 FTP。对于每个请求，会话应保持开放还是关闭？服务器需要保持客户端的状态，还是保存随每个请求一起发送的所有数据？

自定义服务器支持一些简单的功能，如回应和反向发送消息。自定义服务器的另一个特点是，客户端可以发送状态信息，使用另一个调用再次检索它。状态会临时存储在会话状态中。尽管这是一个简单的场景，但我

们知道需要设置它。

TcpServer 示例代码实现为一个控制台应用程序(.NET Core)，利用以下名称空间：

System
System.Collections
System.Collections.Concurrent
System.Linq
System.Net.Sockets
System.Text
System.Threading
System.Threading.Tasks
static TcpServer.CustomProtocol

自定义 TCP 侦听器支持几个请求，如表 23-1 所示。

表 23-1

| 请　　求 | 说　　明 |
|---|---|
| HELO::v1.0 | 启动连接后，这个命令需要发送。其他命令将不被接受 |
| ECHO::message | ECHO 命令向调用者返回消息 |
| REV::message | REV 命令保留消息并返回给调用者 |
| BYE | BYE 命令关闭连接 |
| SET::key=value<br>GET::key | SET 命令设置服务器端状态，可以用 GET 命令检索 |

请求的第一行是一个会话标识符，并带有前缀 ID。它需要与每个请求一起发送，除了 HELO 请求之外。它作为状态标识符使用。

协议的所有常量都在静态类 CustomProtocol 中定义 (代码文件 TcpServer /CustomProtocol.cs)：

```
public static class CustomProtocol
{
  public const string SESSIONID = "ID";
  public const string COMMANDHELO = "HELO";
  public const string COMMANDECHO = "ECO";
  public const string COMMANDREV = "REV";
  public const string COMMANDBYE = "BYE";
  public const string COMMANDSET = "SET";
  public const string COMMANDGET = "GET";
  public const string STATUSOK = "OK";
  public const string STATUSCLOSED = "CLOSED";
  public const string STATUSINVALID = "INV";
  public const string STATUSUNKNOWN = "UNK";
  public const string STATUSNOTFOUND = "NOTFOUND";
  public const string STATUSTIMEOUT = "TIMOUT";
  public const string SEPARATOR = "::";
  public static readonly TimeSpan SessionTimeout = TimeSpan.FromMinutes(2);
}
```

Run()方法(从 Main()方法中调用)启动一个计时器，每分钟清理一次所有的会话状态。Run()方法的主要功能是通过调用 RunServerAsync()方法来启动服务器(代码文件 TcpServer/Program.cs)：

```
static void Main()
{
  var p = new Program();
  p.Run();
}

public void Run()
{
  using (var timer = new Timer(TimerSessionCleanup, null,
    TimeSpan.FromMinutes(1), TimeSpan.FromMinutes(1)))
  {
```

```
    RunServerAsync().Wait();
  }
}
```

对于 TcpListener 类，服务器最重要的部分在 RunServerAsync 方法中。TcpListener 使用 IP 地址和端口号的构造函数实例化，在 IP 地址和端口号上可以访问侦听器。调用 Start 方法，侦听器开始侦听客户端连接。AcceptTcpClientAsync 等待客户端连接。一旦客户端连接，就返回 TcpClient 实例，允许与客户沟通。这个实例传递给 RunClientRequest 方法，以处理请求。

```
private async Task RunServerAsync()
{
  try
  {
    var listener = new TcpListener(IPAddress.Any, portNumber);
    Console.WriteLine($"listener started at port {portNumber}");
    listener.Start();
    while (true)
    {
      Console.WriteLine("waiting for client...");
      TcpClient client = await listener.AcceptTcpClientAsync();
      Task t = RunClientRequest(client);
    }
  }
  catch (Exception ex)
  {
    Console.WriteLine($"Exception of type {ex.GetType().Name}, Message: {ex.Message}");
  }
}
```

为了在客户端上读写，TcpClient 的 GetStream 方法返回 NetworkStream。首先需要读取来自客户机的请求。为此，可以使用 ReadAsync 方法。ReadAsync 方法填充一个字节数组。这个字节数组使用 Encoding 类转换为字符串。收到的信息写入控制台，传递到 ParseRequest 辅助方法。根据 ParseRequest 方法的结果，创建客户端的回应，使用 WriteAsync 方法返回给客户端。

```
private Task RunClientRequestAsync(TcpClient client)
{
  return Task.Run(async () =>
  {
    try
    {
      using (client)
      {
        Console.WriteLine("client connected");
        using (NetworkStream stream = client.GetStream())
        {
          bool completed = false;
          do
          {
            byte[] readBuffer = new byte[1024];
            int read = await stream.ReadAsync(readBuffer, 0, readBuffer.Length);
            string request = Encoding.ASCII.GetString(readBuffer, 0, read);
            Console.WriteLine($"received {request}");

            byte[] writeBuffer = null;
            string response = string.Empty;
            ParseResponse resp = ParseRequest(request, out string sessionId,
              out string result);
            switch (resp)
            {
              case ParseResponse.OK:
                string content = $"{STATUSOK}::{SESSIONID}::{sessionId}";
                if (!string.IsNullOrEmpty(result))
                {
                  content += $"{SEPARATOR}{result}";
                }
                response = $"{STATUSOK}{SEPARATOR}{SESSIONID}{SEPARATOR}" +
                  $"{sessionId}{SEPARATOR}{content}";
                break;
              case ParseResponse.CLOSE:
                response = $"{STATUSCLOSED}";
                completed = true;
                break;
              case ParseResponse.TIMEOUT:
                response = $"{STATUSTIMEOUT}";
```

```
              break;
            case ParseResponse.ERROR:
              response = $"{STATUSINVALID}";
              break;
            default:
              break;
          }
          writeBuffer = Encoding.ASCII.GetBytes(response);
          await stream.WriteAsync(writeBuffer, 0, writeBuffer.Length);
          await stream.FlushAsync();
          Console.WriteLine($"returned {Encoding.ASCII.GetString(
          writeBuffer, 0, writeBuffer.Length)}");
        } while (!completed);
      }
    }
  }
  catch (Exception ex)
  {
    Console.WriteLine($"Exception in client request handling " +
      "of type {ex.GetType().Name}, Message: {ex.Message}");
  }
  Console.WriteLine("client disconnected");
});
}
```

ParseRequest 方法解析请求，并过滤掉会话标识符。server (HELO )的第一个调用是不从客户端传递会话标识符的唯一调用，它是使用 SessionManager 创建的。在第二个和后来的请求中，requestColl[0]必须包含 ID，requestColl[1]必须包含会话标识符。使用这个标识符，如果会话仍然是有效的，TouchSession 方法就更新会话标识符的当前时间。如果无效，就返回超时。对于服务的功能，调用 ProcessRequest 方法：

```
private ParseResponse ParseRequest(string request, out string sessionId,
  out string response)
{
  sessionId = string.Empty;
  response = string.Empty;
  string[] requestColl = request.Split(
  new string[] { SEPARATOR }, StringSplitOptions.RemoveEmptyEntries);
  if (requestColl[0] == COMMANDHELO) // first request
  {
    sessionId = _sessionManager.CreateSession();
  }
  else if (requestColl[0] == SESSIONID) // any other valid request
  {
    sessionId = requestColl[1];
    if (!_sessionManager.TouchSession(sessionId))
    {
      return ParseResponse.TIMEOUT;
    }
    if (requestColl[2] == COMMANDBYE)
    {
      return ParseResponse.CLOSE;
    }
    if (requestColl.Length >= 4)
    {
      response = ProcessRequest(requestColl);
    }
  }
  else
  {
    return ParseResponse.ERROR;
  }
  return ParseResponse.OK;
}
```

ProcessRequest 方法包含一个 switch 语句，来处理不同的请求。这个方法使用 CommandActions 类来回应或反向传递收到的消息。为了存储和检索会话状态，使用 SessionManager：

```
private string ProcessRequest(string[] requestColl)
{
  if (requestColl.Length < 4)
    throw new ArgumentException("invalid length requestColl");

  string sessionId = requestColl[1];
  string response = string.Empty;
  string requestCommand = requestColl[2];
```

```
    string requestAction = requestColl[3];
    switch (requestCommand)
    {
      case COMMANDECHO:
        response = _commandActions.Echo(requestAction);
        break;
      case COMMANDREV:
        response = _commandActions.Reverse(requestAction);
        break;
      case COMMANDSET:
        response = _sessionManager.ParseSessionData(sessionId, requestAction);
        break;
      case COMMANDGET:
        response = $"{_sessionManager.GetSessionData(sessionId, requestAction)}";
        break;
      default:
        response = STATUSUNKNOWN;
        break;
    }
    return response;
  }
```

CommandActions 类定义了简单的方法 Echo 和 Reverse，返回操作字符串，或返回反向发送的字符串 (代码文件 TcpServer/CommandActions.cs)：

```
public class CommandActions
{
  public string Reverse(string action) => string.Join("", action.Reverse());
  public string Echo(string action) => action;
}
```

用 Echo 和 Reverse 方法检查服务器的主要功能后，就要进行会话管理了。服务器上需要一个标识符和上次访问会话的时间，以删除最古老的会话 (代码文件 TcpServer/SessionManager.cs):

```
public struct Session
{
  public string SessionId { get; set; }
  public DateTime LastAccessTime { get; set; }
}
```

SessionManager 包含线程安全的字典，其中存储了所有的会话和会话数据。使用多个客户端时，字典可以在多个线程中同时访问。所以使用名称空间 System.Collections.Concurrent 中线程安全的字典。CreateSession 方法创建一个新的会话，并将其添加到_sessions 字典中：

```
public class SessionManager
{
  private readonly ConcurrentDictionary<string, Session> _sessions =
    new ConcurrentDictionary<string, Session>();
  private readonly ConcurrentDictionary<string, Dictionary<string, string>>
    _sessionData =
      new ConcurrentDictionary<string, Dictionary<string, string>>();

  public string CreateSession()
  {
    string sessionId = Guid.NewGuid().ToString();
    if (_sessions.TryAdd(sessionId,
      new Session
      {
        SessionId = sessionId,
        LastAccessTime = DateTime.UtcNow
      }))
    {
      return sessionId;
    }
    else
    {
      return string.Empty;
    }
  }
  //...
}
```

从计时器线程中，CleanupAllSessions 方法每分钟调用一次，删除最近没有使用的所有会话。该方法又调用 CleanupSession，删除单个会话。客户端发送 BYE 信息时也调用 CleanupSession：

```
public void CleanupAllSessions()
{
  foreach (var session in _sessions)
  {
    if (session.Value.LastAccessTime + SessionTimeout >= DateTime.UtcNow)
    {
      CleanupSession(session.Key);
    }
  }
}

public void CleanupSession(string sessionId)
{
  if (_sessionData.TryRemove(sessionId, out Dictionary<string, string> removed))
  {
    Console.WriteLine($"removed {sessionId} from session data");
  }
  if (_sessions.TryRemove(sessionId, out Session header))
  {
    Console.WriteLine($"removed {sessionId} from sessions");
  }
}
```

TouchSession 方法更新会话的 LastAccessTime，如果会话不再有效，就返回 false：

```
public bool TouchSession(string sessionId)
{
  if (!_sessions.TryGetValue(sessionId, out Session oldHeader))
  {
    return false;
  }
  Session updatedHeader = oldHeader;
  updatedHeader.LastAccessTime = DateTime.UtcNow;
  _sessions.TryUpdate(sessionId, updatedHeader, oldHeader);
  return true;
}
```

为了设置会话数据，需要解析请求。会话数据接收的动作包含由等号分隔的键和值，如 x=42。ParseSessionData 方法解析它，进而调用 SetSessionData 方法：

```
public string ParseSessionData(string sessionId, string requestAction)
{
  string[] sessionData = requestAction.Split('=');
  if (sessionData.Length != 2) return STATUSUNKNOWN;
  string key = sessionData[0];
  string value = sessionData[1];
  SetSessionData(sessionId, key, value);
  return $"{key}={value}";
}
```

SetSessionData 添加或更新字典中的会话状态。GetSessionData 检索值，或返回 NOTFOUND：

```
public string GetSessionData(string sessionId, string key)
{
  if (!_sessionData.TryGetValue(sessionId, out Dictionary<string, string> data))
  {
    data = new Dictionary<string, string>();
    data.Add(key, value);
    _sessionData.TryAdd(sessionId, data);
  }
  else
  {
    if (data.TryGetValue(key, out string val))
    {
      data.Remove(key);
    }
    data.Add(key, value);
  }
}

public string GetSessionData(string sessionId, string key)
{
  if (_sessionData.TryGetValue(sessionId, out Dictionary<string, string> data))
  {
    if (data.TryGetValue(key, out string value))
    {
      return value;
    }
```

```
    }
    return STATUSNOTFOUND;
  }
```

编译侦听器后，可以启动程序。现在，需要一个客户端，以连接到服务器。

### 23.5.3　创建 TCP 客户端

客户端示例是一个 UWP 应用程序 WinAppTCPClient。这个应用程序允许连接到 TCP 服务器，发送自定义协议支持的所有不同命令。

**注意：**
为了使用 UWP 和 TcpClient 类，需要一个支持.NET 标准 2.0 的版本，Fall Creators Update of Windows 10 开始支持它。可下载的代码还包括一个 WPF 示例项目。

应用程序的用户界面如图 23-4 所示。左上部分允许连接到服务器。右上部分的组合框列出了所有命令，Send 按钮向服务器发送命令。在中间部分，显示会话标识符和所发送请求的状态。下部的控件显示服务器接收到的信息，允许清理这些信息。

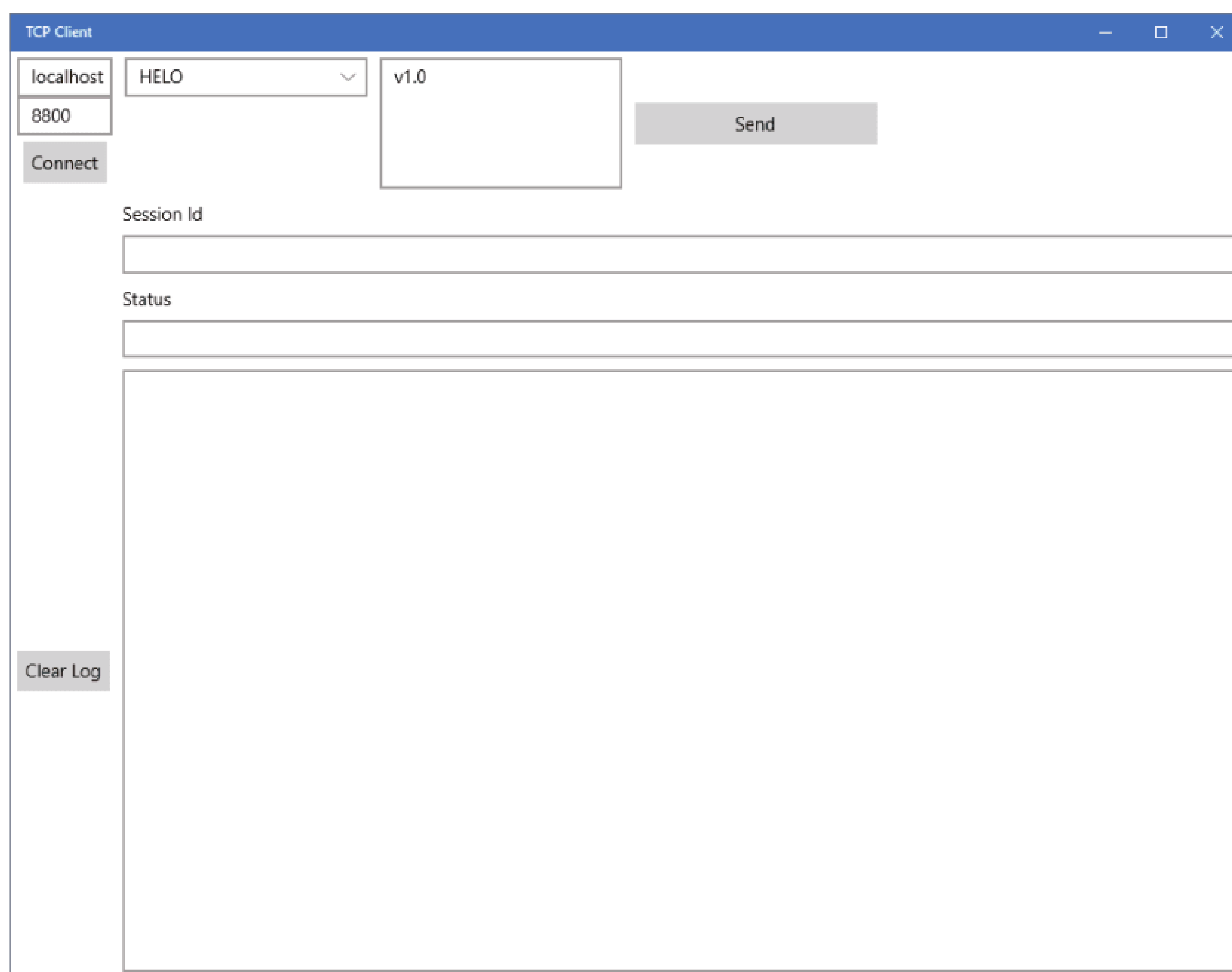



图 23-4

类 CustomProtocolCommand 和 CustomProtocolCommands 用于用户界面中的数据绑定。对于 CustomProtocolCommand，Name 属性显示命令的名称，Action 属性是用户输入的、与命令一起发送的数据。类 CustomProtocolCommands 包含一个绑定到组合框的命令列表(代码文件 WPFAppTcpClient/CustomProtocolCommands.cs)：

```
public class CustomProtocolCommand
{
  public CustomProtocolCommand(string name)
    : this(name, null) { }

  public CustomProtocolCommand(string name, string action)
  {
    Name = name;
    Action = action;
  }
```

```
  public string Name { get; }
  public string Action { get; set; }

  public override string ToString() => Name;
}

public class CustomProtocolCommands : IEnumerable<CustomProtocolCommand>
{
  private readonly List<CustomProtocolCommand> _commands =
    new List<CustomProtocolCommand>();

  public CustomProtocolCommands()
  {
    string[] commands = { "HELO", "BYE", "SET", "GET", "ECO", "REV" };
    foreach (var command in commands)
    {
      _commands.Add(new CustomProtocolCommand(command));
    }
    _commands.Single(c => c.Name == "HELO").Action = "v1.0";
  }

  public IEnumerator<CustomProtocolCommand> GetEnumerator() =>
    _commands.GetEnumerator();

  IEnumerator IEnumerable.GetEnumerator() => _commands.GetEnumerator();
}
```

MainPage 类包含绑定到 XAML 代码的属性和基于用户交互调用的方法。这个类创建 TcpClient 类的一个实例和一些绑定到用户界面的属性(代码文件 WinAppTcpClient/MainPage.xaml.cs)。

```
public partial class MainPage : Page, INotifyPropertyChanged, IDisposable
{
  private TcpClient _client = new TcpClient();
  private readonly CustomProtocolCommands _commands =
    new CustomProtocolCommands();

  public MainWindow() => InitializeComponent();

  private string _remoteHost = "localhost";
  public string RemoteHost
  {
    get => _remoteHost;
    set => SetProperty(ref _remoteHost, value);
  }

  private int _serverPort = 8800;
  public int ServerPort
  {
    get => _serverPort;
    set => SetProperty(ref _serverPort, value);
  }

  private string _sessionId;
  public string SessionId
  {
    get => return _sessionId;
    set => SetProperty(ref _sessionId, value);
  }

  private CustomProtocolCommand _activeCommand;
  public CustomProtocolCommand ActiveCommand
  {
    get => _activeCommand;
    set => SetProperty(ref _activeCommand, value);
  }

  private string _log;
  public string Log
  {
    get => _log;
    set => SetProperty(ref _log, value);
  }

  private string _status;
  public string Status
  {
```

```
    get => return _status;
    set => SetProperty(ref _status, value);
  }
  //...
}
```

当用户单击 Connect 按钮时，调用方法 OnConnect。建立到 TCP 服务器的连接，调用 TcpClient 类的 ConnectAsync 方法。如果连接处于失效模式，且再次调用 OnConnect 方法，就抛出一个 SocketException 异常，其中 ErrorCode 设置为 0x2748。这里使用 C#异常过滤器来处理 SocketException，创建一个新的 TcpClient，因此再次调用 OnConnect 可能会成功(代码文件 WinAppTcpClient/MainPage.xaml.cs)：

```
private async void OnConnect(object sender, RoutedEventArgs e)
{
  try
  {
    await _client.ConnectAsync(RemoteHost, ServerPort);
  }
  catch (SocketException ex) when (ex.ErrorCode == 0x2748)
  {
    _client.Dispose();
    _client = new TcpClient();
    await new MessageDialog("please retry connect").ShowAsync();
  }
  catch (Exception ex)
  {
    await new MessageDialog(ex.Message).ShowAsync();
  }
}
```

请求发送到 TCP 服务器是由 OnSendCommand 方法处理的。这里的代码非常类似于服务器上的收发代码。GetStream 方法返回一个 NetworkStream，这用于把(WriteAsync)数据写入服务器，从服务器中读取(ReadAsync)数据(代码文件 WinAppTcpClient/MainPage.xaml.cs)：

```
private async void OnSendCommand(object sender, RoutedEventArgs e)
{
  try
  {
    if (!VerifyIsConnected()) return;
    NetworkStream stream = _client.GetStream();
    byte[] writeBuffer = Encoding.ASCII.GetBytes(GetCommand());
    await stream.WriteAsync(writeBuffer, 0, writeBuffer.Length);
    await stream.FlushAsync();
    byte[] readBuffer = new byte[1024];
    int read = await stream.ReadAsync(readBuffer, 0, readBuffer.Length);
    string messageRead = Encoding.ASCII.GetString(readBuffer, 0, read);
    Log += messageRead + Environment.NewLine;
    ParseMessage(messageRead);
  }
  catch (Exception ex)
  {
    await new MessageDialog(ex.Message).ShowAsync();
  }
}
```

为了建立可以发送到服务器的数据，从 OnSendCommand 内部调用 GetCommand 方法。GetCommand 又调用方法 GetSessionHeader 来建立会话标识符，然后提取 ActiveCommand 属性(其类型是 CustomProtocolCommand)，其中包含选中的命令名称和输入的数据：

```
private string GetCommand() =>
  $"{GetSessionHeader()}{ActiveCommand?.Name}::{ActiveCommand?.Action}";

private string GetSessionHeader()
{
  if (string.IsNullOrEmpty(SessionId)) return string.Empty;
  return $"ID::{SessionId}::";
}
```

从服务器接收数据后使用 ParseMessage 方法。这个方法拆分消息以设置 Status 和 SessionId 属性：

```
private void ParseMessage(string message)
{
  if (string.IsNullOrEmpty(message)) return;
  string[] messageColl = message.Split(
```

```
      new string[] { "::" }, StringSplitOptions.RemoveEmptyEntries);
    Status = messageColl[0];
    SessionId = GetSessionId(messageColl);
  }
```

运行应用程序时，可以连接到服务器，选择命令，设置回应和反向发送的值，查看来自服务器的所有消息，如图 23-5 所示。

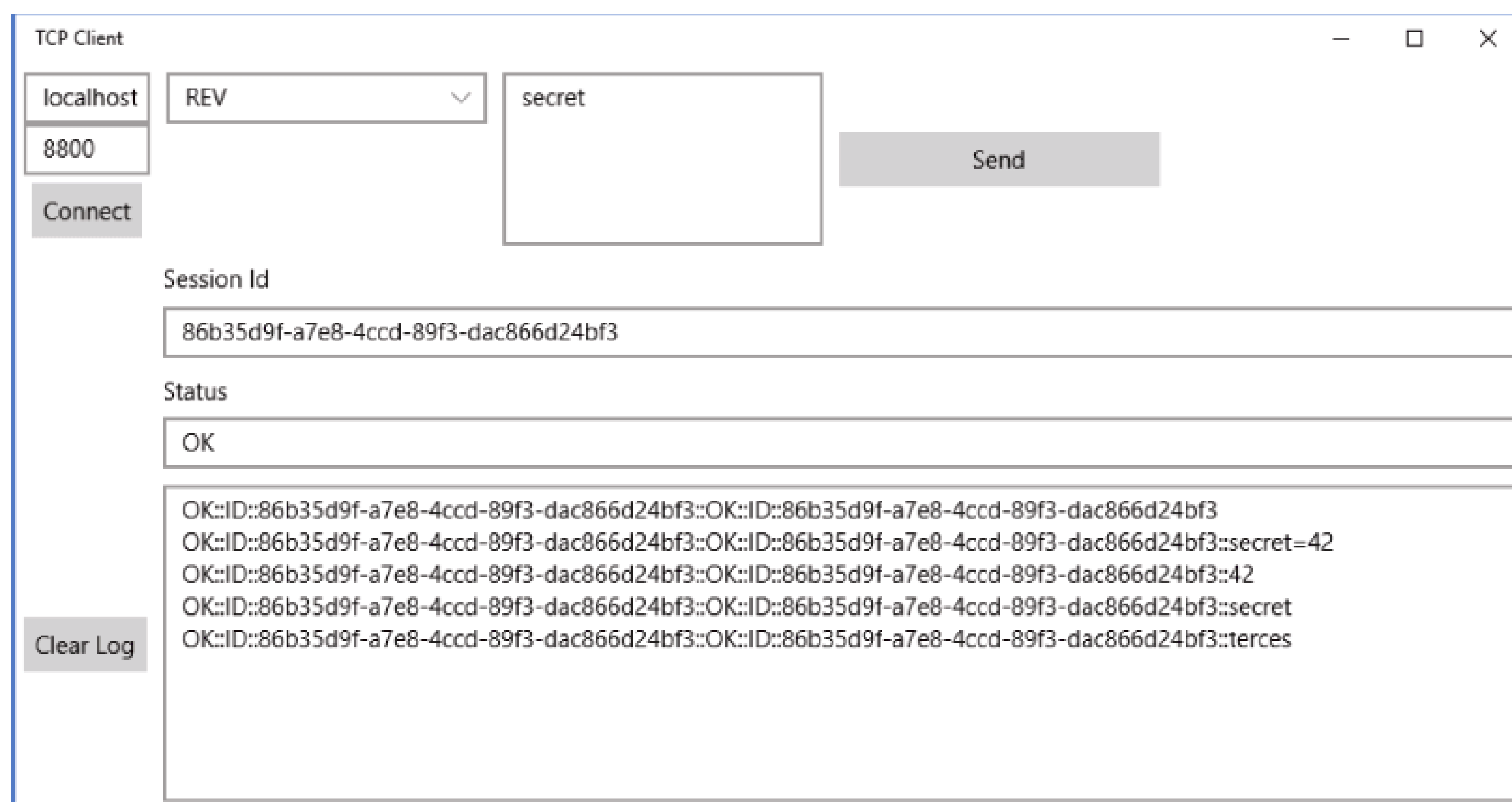



图 23-5

### 23.5.4 TCP 和 UDP

本节要介绍的另一个协议是 UDP(用户数据报协议)。UDP 是一个几乎没有开销的简单协议。在使用 TCP 发送和接收数据之前，需要建立连接。而这对于 UDP 是没有必要的。使用 UDP 只需要开始发送或接收。当然，这意味着 UDP 开销低于 TCP，但也更不可靠。当使用 UDP 发送数据时，接收这些数据时就没有得到信息。UDP 经常用于速度和性能需求大于可靠性要求的情形，例如视频流。UDP 还可以把消息广播到一组节点。相反，TCP 提供了许多功能来确保数据的传输，它还提供了错误校正以及当数据丢失或数据包损坏时重新传输它们的功能。最后，TCP 可缓冲传入和传出的数据，还保证在传输过程中，在把数据包传送给应用程序之前重组杂乱的一系列数据包。即使有一些额外的开销，TCP 仍是在 Internet 上使用最广泛的协议，因为它有非常高的可靠性。

## 23.6 使用 UDP

为了演示 UDP，创建两个控制台应用程序(包)项目，显示 UDP 的各种特性：直接将数据发送到主机，在本地网络上把数据广播到所有主机上，把数据多播到属于同一个组的一组节点上。

UdpSender 和 UdpReceiver 项目使用以下名称空间：

System
System.Linq
System.Net
System.Net.Sockets
System.Text
System.Threading.Tasks

### 23.6.1 建立 UDP 接收器

从接收应用程序开始。该应用程序使用命令行参数来控制应用程序的不同功能。所需的命令行参数是 -p，它指定接收器可以接收数据的端口号。可选参数-g 与一个组地址用于多播。ParseCommandLine 方

法解析命令行参数，并将结果放入变量 port 和 groupAddress 中(代码文件 UdpReceiver/Program.cs)：

```
static async Task Main(string[] args)
{
  if (!ParseCommandLine(args, out int port, out string groupAddress))
  {
    ShowUsage();
    return;
  }
  await ReaderAsync(port, groupAddress);
  Console.ReadLine();
}

private static void ShowUsage()
{
  Console.WriteLine("Usage: UdpReceiver -p port [-g groupaddress]");
}
```

Reader 方法使用在程序参数中传入的端口号创建一个 UdpClient 对象。ReceiveAsync 方法等到一些数据的到来。这些数据可以使用 UdpReceiveResult 和 Buffer 属性找到。数据编码为字符串后，写入控制台，继续循环，等待下一个要接收的数据：

```
private static async Task ReaderAsync(int port, string groupAddress)
{
  using (var client = new UdpClient(port))
  {
    if (groupAddress != null)
    {
      client.JoinMulticastGroup(IPAddress.Parse(groupAddress));
      Console.WriteLine(
        $"joining the multicast group {IPAddress.Parse(groupAddress)}");
    }

    bool completed = false;
    do
    {
      Console.WriteLine("starting the receiver");
      UdpReceiveResult result = await client.ReceiveAsync();
      byte[] datagram = result.Buffer;
      string received = Encoding.UTF8.GetString(datagram);
      Console.WriteLine($"received {received}");
      if (received == "bye")
      {
        completed = true;
      }
    } while (!completed);
    Console.WriteLine("receiver closing");
    if (groupAddress != null)
    {
      client.DropMulticastGroup(IPAddress.Parse(groupAddress));
    }
  }
}
```

启动应用程序时，它等待发送方发送数据。目前，忽略多播组，只使用参数和端口号，因为多播在创建发送器后讨论。

## 23.6.2　创建 UDP 发送器

UDP 发送器应用程序还允许通过命令行选项进行配置。它比接收应用程序有更多的选项。除了命令行参数 - p 指定端口号之外，发送方还允许使用- b 在本地网络中广播到所有节点，使用- h 识别特定的主机，使用-g 指定一个组，使用-ipv6 表明应该使用 IPv6 取代 IPv4 (代码文件 UdpSender/Program.cs):

```
static async Task Main(string[] args)
{
  if (!ParseCommandLine(args, out int port, out string hostname, out bool broadcast,
    out string groupAddress, out bool ipv6))
  {
    ShowUsage();
    Console.ReadLine();
    return;
  }
```

```
    IPEndpoint endpoint = await GetIPEndPointAsync(port, hostname, broadcast,
      groupAddress, ipv6);
    await SenderAsync(endpoint, broadcast, groupAddress);
    Console.WriteLine("Press return to exit...");
    Console.ReadLine();
  }

  private static void ShowUsage()
  {
    Console.WriteLine("Usage: UdpSender -p port [-g groupaddress | -b | -h hostname] " +
      "[-ipv6]");
    Console.WriteLine("\t-p port number\tEnter a port number for the sender");
    Console.WriteLine("\t-g group address\tGroup address in the range 224.0.0.0 " +
      "to 239.255.255.255");
    Console.WriteLine("\t-b\tFor a broadcast");
    Console.WriteLine("\t-h hostname\tUse the hostname option if the message should " +
      "be sent to a single host");
  }
```

发送数据时，需要一个 IPEndPoint。根据程序参数，以不同的方式创建它。对于广播，IPv4 定义了从 IPAddress.Broadcast 返回的地址 255.255.255.255。没有用于广播的 IPv6 地址，因为 IPv6 不支持广播。IPv6 用多播替代广播。多播也添加到 IPv4 中。

传递主机名时，主机名使用 DNS 查找功能和 Dns 类来解析。GetHostEntryAsync 方法返回一个 IPHostEntry，其中 IPAddress 可以从 AddressList 属性中检索。根据使用 IPv4 还是 IPv6，从这个列表中提取不同的 IPAddress。根据网络环境，只有一个地址类型是有效的。如果把一个组地址传递给方法，就使用 IPAddress.Parse 解析地址：

```
  public static async Task<IPEndPoint> GetIPEndPoint(int port, string hostName,
  bool broadcast, string groupAddress, bool ipv6)
  {
    IPEndPoint endpoint = null;
    try
    {
      if (broadcast)
      {
        endpoint = new IPEndPoint(IPAddress.Broadcast, port);
      }
      else if (hostName != null)
      {
        IPHostEntry hostEntry = await Dns.GetHostEntryAsync(hostName);
        IPAddress address = null;
        if (ipv6)
        {
          address = hostEntry.AddressList.Where(
              a => a.AddressFamily == AddressFamily.InterNetworkV6)
            .FirstOrDefault();
        }
        else
        {
           address = hostEntry.AddressList.Where(
              a => a.AddressFamily == AddressFamily.InterNetwork)
            .FirstOrDefault();
        }
        if (address == null)
        {
          Func<string> ipversion = () => ipv6 ? "IPv6" : "IPv4";
          Console.WriteLine($"no {ipversion()} address for {hostName}");
          return null;
        }
        endpoint = new IPEndPoint(address, port);
      }
      else if (groupAddress != null)
      {
        endpoint = new IPEndPoint(IPAddress.Parse(groupAddress), port);
      }
      else
      {
        throw new InvalidOperationException($"{nameof(hostName)}, "
          + "{nameof(broadcast)}, or {nameof(groupAddress)} must be set");
      }
    }
    catch (SocketException ex)
    {
      Console.WriteLine(ex.Message);
```

```
    }
    return endpoint;
  }
```

现在，关于 UDP 协议，讨论发送器最重要的部分。在创建一个 UdpClient 实例，并将字符串转换为字节数组后，就使用 SendAsync 方法发送数据。请注意接收器不需要侦听，发送方也不需要连接。UDP 是很简单的。然而，如果发送方把数据发送到未知的地方——无人接收数据，也不会得到任何错误消息：

```
private async Task Sender(IPEndpoint endpoint, bool broadcast,
  string groupAddress)
{
  try
  {
    string localhost = Dns.GetHostName();
    using (var client = new UdpClient())
    {
      client.EnableBroadcast = broadcast;
      if (groupAddress != null)
      {
        client.JoinMulticastGroup(IPAddress.Parse(groupAddress));
      }
      bool completed = false;
      do
      {
        Console.WriteLine("Enter a message or bye to exit");
        string input = Console.ReadLine();
        Console.WriteLine();
        completed = input == "bye";
        byte[] datagram = Encoding.UTF8.GetBytes($"{input} from {localhost}");
        int sent = await client.SendAsync(datagram, datagram.Length, endpoint);
      } while (!completed);

      if (groupAddress != null)
      {
        client.DropMulticastGroup(IPAddress.Parse(groupAddress));
      }
    }
  }
  catch (SocketException ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

现在可以用如下选项启动接收器：

```
-p 9400
```

用如下选项启动发送器：

```
-p 9400 -h localhost
```

可以在发送器中输入数据，发送到接收器。如果停止接收器，就可以继续发送，而不会检测到任何错误。也可以尝试使用主机名而不是 localhost，并在另一个系统上运行接收器。

在发送器中，可以添加-b 选项，删除主机名，给在同一个网络上侦听端口 9400 的所有节点发送广播：

```
-p 9400 -b
```

请注意广播不跨越大多数路由器，当然不能在互联网上使用广播。这种情况和多播不同，参见下面的讨论。

### 23.6.3 使用多播

广播不跨越路由器，但多播可以跨越。多播用于将消息发送到一组系统上——所有节点都属于同一个组。在 IPv4 中，为使用多播保留了特定的 IP 地址。地址是从 224.0.0.0 到 239.255.255.253。这些地址中的许多都保留给具体的协议，例如用于路由器，但 239.0.0.0/8 可以私下在组织中使用。这非常类似于 IPv6，它为不同的路由协议保留了著名的 IPv6 多播地址。地址 f::/16 是组织中的本地地址，地址 ffxe::/16 有全局作用域，可以在公共互联网上路由。

对于使用多播的发送器或接收器，必须通过调用 UdpClient 的 JoinMulticastGroup 方法来加入一个多播组：

```
client.JoinMulticastGroup(IPAddress.Parse(groupAddress));
```

为了再次退出该组，可以调用方法 DropMulticastGroup：

```
client.DropMulticastGroup(IPAddress.Parse(groupAddress));
```

用如下选项启动接收器和发送器：

```
-p 9400 -g 230.0.0.1
```

它们都属于同一个组，多播在进行。和广播一样，可以启动多个接收器和多个发送器。接收器将接收来自每个接收器的几乎所有消息。

# 23.7 使用套接字

HTTP 协议基于 TCP，因此 HttpXX 类在 TcpXX 类上提供了一个抽象层。然而 TcpXX 类提供了更多的控制。使用套接字，甚至可以获得比 TcpXX 或 UdpXX 类更多的控制。通过套接字，可以使用不同的协议，不仅是基于 TCP 或 UDP 的协议，还可以创建自己的协议。更重要的是，可以更多地控制基于 TCP 或 UDP 的协议。

SocketServerSender 和 SocketClient 项目实现为控制台应用程序(包)，使用如下名称空间：

System
System.Linq
System.IO
System.Net
System.Net.Sockets
System.Text
System.Threading
System.Threading.Tasks

## 23.7.1 使用套接字创建侦听器

首先用一个服务器侦听传入的请求。服务器需要一个用程序参数传入的端口号。之后，就调用 Listener 方法(代码文件 SocketServer/Program.cs)：

```
static void Main(string[] args)
{
  if (args.Length != 1)
  {
    ShowUsage();
    return;
  }

  if (!int.TryParse(args[0], out int port))
  {
    ShowUsage();
    return;
  }

  Listener(port);
  Console.ReadLine();
}

private void ShowUsage()
{
  Console.WriteLine("SocketServer port");
}
```

对套接字最重要的代码在下面的代码片段中。侦听器创建一个新的 Socket 对象。给构造函数提供 AddressFamily、SocketType 和 ProtocolType。AddressFamily 是一个大型枚举，提供了许多不同的网络。例如 DECnet(Digital Equipment 在 1975 年发布它，主要用作 PDP-11 系统之间的网络通信)；Banyan VINES(用于连接客户机)；当然还有用于 IPv4 的 InternetWork 和用于 IPv6 的 InternetWorkV6。如前所述，可以为大量网络协议使用套接字。第二个参数 SocketType 指定套接字的类型。例如用于 TCP 的 Stream、用于 UDP 的 Dgram 或用于

原始套接字的 Raw。第三个参数是用于 ProtocolType 的枚举。例如 IP、Ucmp、Udp、IPv6 和 Raw。所选的设置需要匹配。例如，使用 TCP 与 IPv4，地址系列就必须是 InterNetwork、套接字类型 Stream、协议类型 Tcp。要使用 IPv4 创建一个 UDP 通信，地址系列就需要设置为 InterNetwork、套接字类型 Dgram 和协议类型 Udp。

```
public static void Listener(int port)
{
  var listener = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
  ProtocolType.Tcp);
  listener.ReceiveTimeout = 5000; // receive timout 5 seconds
  listener.SendTimeout = 5000; // send timeout 5 seconds
  //...
```

从构造函数返回的侦听器套接字绑定到 IP 地址和端口号上。在示例代码中，侦听器绑定到所有本地 IPv4 地址上，端口号用参数指定。调用 Listen 方法，启动套接字的侦听模式。套接字现在可以接受传入的连接请求。用 Listen 方法指定参数，定义了服务器的缓冲区队列的大小——在处理连接之前，可以同时连接多少客户端：

```
public static void Listener(int port)
{
  //...
  listener.Bind(new IPEndPoint(IPAddress.Any, port));
  listener.Listen(backlog: 15);
  Console.WriteLine($"listener started on port {port}");
  //...
```

等待客户端连接在 Socket 类的方法 Accept 中进行。这个方法阻塞线程，直到客户机连接为止。客户端连接后，需要再次调用这个方法，来满足其他客户端的请求；所以在 while 循环中调用此方法。为了进行侦听，启动一个单独的任务，该任务可以在调用线程中取消。在方法 CommunicateWithClientUsingSocketAsync 中执行使用套接字读写的任务。这个方法接收绑定到客户端的 Socket 实例，进行读写：

```
public static void Listener(int port)
{
  //...
  var cts = new CancellationTokenSource();
  var tf = new TaskFactory(TaskCreationOptions.LongRunning,
  TaskContinuationOptions.None);
  tf.StartNew(() => // listener task
  {
    Console.WriteLine("listener task started");
    while (true)
    {
      if (cts.Token.IsCancellationRequested)
      {
        cts.Token.ThrowIfCancellationRequested();
        break;
      }
      Console.WriteLine("waiting for accept");

      Socket client = listener.Accept();
      if (!client.Connected)
      {
        Console.WriteLine("not connected");
        continue;
      }
      Console.WriteLine($"client connected local address " +
        $"{((IPEndPoint)client.LocalEndPoint).Address} and port " +
        $"{((IPEndPoint)client.LocalEndPoint).Port}, remote address " +
        $"{((IPEndPoint)client.RemoteEndPoint).Address} and port " +
        $"{((IPEndPoint)client.RemoteEndPoint).Port}");
      Task t = CommunicateWithClientUsingSocketAsync(client);
    }
    listener.Dispose();
    Console.WriteLine("Listener task closing");
  }, cts.Token);
  Console.WriteLine("Press return to exit");
  Console.ReadLine();
  cts.Cancel();
}
```

为了与客户端沟通，创建一个新任务。这会释放侦听器任务，立即进行下一次迭代，等待下一个客户端连接。Socket 类的 Receive 方法接受一个缓冲，其中的数据和标志可以读取，用于套接字。这个字节数组转换为字符串，使用 Send 方法，连同一个小变化一起发送回客户机：

```
private static Task CommunicateWithClientUsingSocketAsync(Socket socket)
{
  return Task.Run(() =>
  {
    try
    {
      using (socket)
      {
        bool completed = false;
        do
        {
          byte[] readBuffer = new byte[1024];
          int read = socket.Receive(readBuffer, 0, 1024, SocketFlags.None);
          string fromClient = Encoding.UTF8.GetString(readBuffer, 0, read);
          Console.WriteLine($"read {read} bytes: {fromClient}");
          if (string.Compare(fromClient, "shutdown", ignoreCase: true) == 0)
          {
            completed = true;
          }
          byte[] writeBuffer = Encoding.UTF8.GetBytes($"echo {fromClient}");
          int send = socket.Send(writeBuffer);
          Console.WriteLine($"sent {send} bytes");
        } while (!completed);
      }
      Console.WriteLine("closed stream and client socket");
    }
    catch (Exception ex)
    {
      Console.WriteLine(ex.Message);
    }
  });
}
```

服务器已经准备好了。然而，下面看看通过扩展抽象级别读写通信信息的不同方式。

## 23.7.2 使用 NetworkStream 和套接字

前面使用了 NetworkStream 类、TcpClient 和 TcpListener 类。NetworkStream 构造函数允许传递 Socket，所以可以使用流方法 Read 和 Write 替代套接字的 Send 和 Receive 方法。在 NetworkStream 的构造函数中，可以定义流是否应该拥有套接字。如这段代码所示，如果流拥有套接字，就在关闭流时关闭套接字(代码文件SocketServer/Program.cs)：

```
private static async Task CommunicateWithClientUsingNetworkStreamAsync(
  Socket socket)
{
  try
  {
    using (var stream = new NetworkStream(socket, ownsSocket: true))
    {
      bool completed = false;
      do
      {
        byte[] readBuffer = new byte[1024];
        int read = await stream.ReadAsync(readBuffer, 0, 1024);
        string fromClient = Encoding.UTF8.GetString(readBuffer, 0, read);
        Console.WriteLine($"read {read} bytes: {fromClient}");
        if (string.Compare(fromClient, "shutdown", ignoreCase: true) == 0)
        {
          completed = true;
        }
        byte[] writeBuffer = Encoding.UTF8.GetBytes($"echo {fromClient}");
        await stream.WriteAsync(writeBuffer, 0, writeBuffer.Length);
      } while (!completed);
    }
    Console.WriteLine("closed stream and client socket");
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

要在代码示例中使用这个方法，需要更改 Listener 方法，调用 CommunicateWithClientUsingNetworkStreamAsync

而不是 CommunicateWithClientUsingSocketAsync 方法。

### 23.7.3　通过套接字使用读取器和写入器

下面再添加一个抽象层。因为 NetworkStream 派生于 Stream 类，还可以使用读取器和写入器访问套接字。只需要注意读取器和写入器的生存期。调用读取器和写入器的 Dispose 方法，还会销毁底层的流。所以要选择 StreamReader 和 StreamWriter 的构造函数，其中 leaveOption 参数可以设置为 true。之后，在销毁读取器和写入器时，就不会销毁底层的流了。NetworkStream 在外层 using 语句的最后销毁，这又会关闭套接字，因为它拥有套接字。还有另一个方面需要注意：通过套接字使用写入器时，默认情况下，写入器不刷新数据，所以它们保存在缓存中，直到缓存已满。使用网络流，可能需要更快的回应。这里可以把 AutoFlush 属性设置为 true(也可以调用 FlushAsync 方法)：

```
public static async Task CommunicateWithClientUsingReadersAndWritersAsync(
  Socket socket)
{
  try
  {
    using (var stream = new NetworkStream(socket, ownsSocket: true))
    using (var reader = new StreamReader(stream, Encoding.UTF8, false,
      8192, leaveOpen: true))
    using (var writer = new StreamWriter(stream, Encoding.UTF8,
      8192, leaveOpen: true))
    {
      writer.AutoFlush = true;
      bool completed = false;
      do
      {
        string fromClient = await reader.ReadLineAsync();
        Console.WriteLine($"read {fromClient}");
        if (string.Compare(fromClient, "shutdown", ignoreCase: true) == 0)
        {
          completed = true;
        }
          await writer.WriteLineAsync($"echo {fromClient}");
      } while (!completed);
    }
    Console.WriteLine("closed stream and client socket");
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

要在代码示例中使用这个方法，需要更改 Listener 方法来调用方法 CommunicateWithClientUsing-ReadersAndWritersAsync，而不是方法 CommunicateWithClientUsingSocketAsync。

**注意：**
流、读取器和写入器参见第 22 章。

### 23.7.4　使用套接字实现接收器

接收方应用程序 SocketClient 也实现为一个控制台应用程序(包)。通过命令行参数，需要传递服务器的主机名和端口号。成功解析命令行后，调用方法 SendAndReceive 与服务器通信(代码文件 SocketClient/Program.cs)：

```
static async Task Main(string[] args)
{
  if (args.Length != 2)
  {
    ShowUsage();
    return;
  }

  string hostName = args[0];
  if (!int.TryParse(args[1], out int port))
  {
```

```
      ShowUsage();
      return;
    }
    Console.WriteLine("press return when the server is started");
    Console.ReadLine();
    await SendAndReceiveAsync(hostName, port);
    Console.ReadLine();
  }

  private static void ShowUsage()
  {
    Console.WriteLine("Usage: SocketClient server port");
  }
```

SendAndReceive 方法使用 DNS 名称解析，从主机名中获得 IPHostEntry。这个 IPHostEntry 用来得到主机的 IPv4 地址。创建 Socket 实例后(其方式与为服务器创建代码相同)， Connect 方法使用该地址连接到服务器。连接完成后，调用 Sender 和 Receiver 方法，创建不同的任务，这允许同时运行这些方法。接收方客户端可以同时读写服务器：

```
public static async Task SendAndReceiveAsync(string hostName, int port)
{
  try
  {
    IPHostEntry ipHost = await Dns.GetHostEntryAsync(hostName);
    IPAddress ipAddress = ipHost.AddressList.Where(
      address => address.AddressFamily == AddressFamily.InterNetwork).First();
    if (ipAddress == null)
    {
      Console.WriteLine("no IPv4 address");
      return;
    }

    using (var client = new Socket(AddressFamily.InterNetwork,
      SocketType.Stream, ProtocolType.Tcp))
    {
      client.Connect(ipAddress, port);
      Console.WriteLine("client successfully connected");
      var stream = new NetworkStream(client);
      var cts = new CancellationTokenSource();
      Task tSender = SenderAsync(stream, cts);
      Task tReceiver = ReceiverAsync(stream, cts.Token);
      await Task.WhenAll(tSender, tReceiver);
    }
  }
  catch (SocketException ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

**注意：**

如果改变地址列表的过滤方式，得到一个 IPv6 地址，而不是 IPv4 地址，则还需要改变 Socket 调用，为 IPv6 地址系列创建一个套接字。

Sender 方法要求用户输入数据，并使用 WriteAsync 方法将这些数据发送到网络流。Receiver 方法用 ReadAsync 方法接收流中的数据。当用户进入终止字符串时，通过 CancellationToken 从 Sender 任务中发送取消信息：

```
public static async Task SenderAsync(NetworkStream stream,
  CancellationTokenSource cts)
{
  Console.WriteLine("Sender task");
  while (true)
  {
    Console.WriteLine("enter a string to send, shutdown to exit");
    string line = Console.ReadLine();
    byte[] buffer = Encoding.UTF8.GetBytes($"{line}\r\n");
    await stream.WriteAsync(buffer, 0, buffer.Length);
    await stream.FlushAsync();
    if (string.Compare(line, "shutdown", ignoreCase: true) == 0)
    {
```

```
            cts.Cancel();
            Console.WriteLine("sender task closes");
            break;
          }
        }
      }

      private const int ReadBufferSize = 1024;

      public static async Task ReceiverAsync(NetworkStream stream,
        CancellationToken token)
      {
        try
        {
          stream.ReadTimeout = 5000;
          Console.WriteLine("Receiver task");
          byte[] readBuffer = new byte[ReadBufferSize];
          while (true)
          {
            Array.Clear(readBuffer, 0, ReadBufferSize);
            int read = await stream.ReadAsync(readBuffer, 0, ReadBufferSize, token);
            string receivedLine = Encoding.UTF8.GetString(readBuffer, 0, read);
            Console.WriteLine($"received {receivedLine}");
          }
        }
        catch (OperationCanceledException ex)
        {
          Console.WriteLine(ex.Message);
        }
      }
```

运行客户端和服务器，可以看到通过 TCP 的通信。

**注意：**

示例代码实现了一个 TCP 客户机和服务器。TCP 需要一个连接，才能发送和接收数据；为此要调用 Connect 方法。对于 UDP，也可以调用 Connect 连接方法，但它不建立连接。使用 UDP 时，不是调用 Connect 方法，而可以使用 SendTo 和 ReceiveFrom 方法代替。这些方法需要一个 EndPoint 参数，在发送和接收时定义端点。

**注意：**

取消标记参见第 21 章。

## 23.8　小结

本章回顾了 System.Net 名称空间中用于网络通信的.NET Framework 类。从中可了解到，某些.NET 基类可处理在网络和 Internet 上打开的客户端连接，如何给服务器发送请求和从服务器接收响应

作为经验规则，在使用 System.Net 名称空间中的类编程时，应尽可能一直使用最通用的类。例如，使用 TCPClient 类代替 Socket 类，可以把代码与许多低级套接字细节分离开来。更进一步，HttpClient 类是利用 HTTP 协议的一种简单方式。

本书更多地讨论网络，而不是本章提到的核心网络功能。第 32 章将介绍 ASP.NET Web API，它使用 HTTP 协议提供服务。网上附加第 3 章探讨 WebHooks 和 SignalR，这两个技术提供了事件驱动的通信。

下一章讨论安全性，说明 CryptoStream 如何用于加密流，无论流是用于文件还是联网。

# 第24章 安　全　性

**本章要点**

- 身份验证和授权
- 创建和验证签名
- 保护数据交换
- 签名和散列
- 数据保护
- 资源的访问控制
- Web 安全性

**本章源代码下载地址(wrox.com)：**

打开 www.wrox.com 的 Download Code 选项卡可下载本章源代码。源代码也可以在 Security 目录的 https://github.com/ProfessionalCSharp/ProfessionalCSharp7 中找到。本章代码分为以下几个主要的示例文件：

- WindowsPrincipal
- SigningDemo
- SecureTransfer
- RSASample
- DataProtection
- UserSecretsSample
- FileAccessControl
- WebApplicationSecurity

## 24.1 概述

为了确保应用程序的安全，安全性有几个重要方面需要考虑。一是应用程序的用户，访问应用程序的是一个真正的用户，还是伪装成用户的某个人？如何确定这个用户是可以信任的？如本章所述，确保应用程序安全的用户方面是一个两阶段过程：用户首先需要进行身份验证，再进行授权，以验证该用户是否可以使用需要的资源。

对于在网络上存储或发送的数据呢？例如，有人可以通过网络嗅探器访问这些数据吗？这里，数据的加密很重要。一些技术，如 Windows Communication Foundation(WCF)，通过简单的配置提供了加密功能，所以可以看到后台执行了什么操作。

另一方面是应用程序本身。如果应用程序驻留在 Web 提供程序上，如何禁止应用程序执行对服务器有害的操作？

本章将讨论.NET 中有助于管理安全性的一些特性，其中包括.NET 如何避开恶意代码、如何管理安全性策略，以及如何通过编程访问安全子系统等。

本章还会讨论保护 Web 应用程序时需要注意的问题。

# 24.2　验证用户信息

安全性的两个基本支柱是身份验证和授权。身份验证是标识用户的过程，授权在验证了所标识用户是否可以访问特定资源之后进行。本节介绍如何使用标识符和 principals 获得用户的信息。

## 24.2.1　使用 Windows 标识

使用标识可以验证运行应用程序的用户。WindowsIndentity 类表示一个 Windows 用户。如果没有用 Windows 账户标识用户，也可以使用实现了 IIdentity 接口的其他类。通过这个接口可以访问用户名、该用户是否通过身份验证，以及验证类型等信息。

principal 是一个包含用户的标识和用户的所属角色的对象。IPrincipal 接口定义了 Identity 属性和 IsInRole()方法，Identity 属性返回 IIdentity 对象；在 IsInRole()方法中，可以验证用户是否是指定角色的一个成员。角色是有相同安全权限的用户集合，同时它是用户的管理单元。角色可以是 Windows 组或自己定义的一个字符串集合。

.NET 中的 Principal 类有 WindowsPrincipal、GenericPrincipal 和 RolePrincipal。从.NET 4.5 开始，这些 Principal 类型派生于基类 ClaimsPrincipal。还可以创建实现了 IPrincipal 接口或派生于 ClaimsPrincipal 的自定义 Principal 类。

示例代码仅运行在 Windows 上，使用如下依赖项和名称空间：

**依赖项**

System.Security.Principal.Windows

**名称空间**

System

System.Collections.Generic

System.Security.Claims

System.Security.Principal

下面创建一个控制台应用程序(.NET Core)，它可以访问某个应用程序中的主体，以便允许用户访问底层的 Windows 账户。这里需要导入 System.Security.Principal 和 System.Security.Claims 名称空间。Main()方法调用方法 ShowIdentityInformation()把 WindowsIdentity 的信息写到控制台，调用 ShowPrincipal 写入可用于 principals 的额外信息，调用 ShowClaims 写入声称信息(代码文件 WindowsPrincipal/Program.cs)：

```
static void Main()
{
  WindowsIdentity identity = ShowIdentityInformation();
  WindowsPrincipal principal = ShowPrincipal(identity);
  ShowClaims(principal.Claims);
}
```

ShowIdentityInformation()方法通过调用 WindowsIdentity 的静态方法 GetCurrent，创建一个 WindowsIdentity 对象，并访问其属性，来显示身份类型、名称、身份验证类型和其他值(代码文件 WindowsPrincipal/Program.cs)：

```
public static WindowsIdentity ShowIdentityInformation()
{
```

```
  WindowsIdentity identity = WindowsIdentity.GetCurrent();
  if (identity == null)
  {
    Console.WriteLine("not a Windows Identity");
    return null;
  }
  Console.WriteLine($"IdentityType: {identity}");
  Console.WriteLine($"Name: {identity.Name}");
  Console.WriteLine($"Authenticated: {identity.IsAuthenticated}");
  Console.WriteLine($"Authentication Type: {identity.AuthenticationType}");
  Console.WriteLine($"Anonymous? {identity.IsAnonymous}");
  Console.WriteLine($"Access Token: " +
    $"{identity.AccessToken.DangerousGetHandle()}");
  Console.WriteLine();
  return identity;
}
```

所有的标识类，例如 WindowsIdentity，都实现了 IIdentity 接口，该接口包含 3 个属性(AuthenticationType、IsAuthenticated 和 Name)，便于所有的派生标识类实现它们。WindowsIdentity 的其他属性都专用于这种标识。

运行应用程序，信息如以下代码片段所示。身份验证类型显示 CloudAP，因为使用 Microsoft Live 账户登录到系统。如果使用 Active Directory，Active Directory 就显示在验证类型中：

```
IdentityType: System.Security.Principal.WindowsIdentity
Name: THEROCKS\Christian
Authenticated: True
Authentication Type: CloudAP
Anonymous? False
Access Token: 564
```

### 24.2.2 Windows Principal

principal 包含一个标识，提供额外的信息，比如用户所属的角色。principal 实现了 IPrincipal 接口，提供了方法 IsInRole 和 Identity 属性。在 Windows 中，用户所属的所有 Windows 组映射到角色。重载 IsInRole 方法，以接受安全标识符、角色字符串或 WindowsBuiltInRole 枚举的值。示例代码验证用户是否属于内置的角色 User 和 Administrator (代码文件 WindowsPrincipal/Program.cs)：

```
public static WindowsPrincipal ShowPrincipal(WindowsIdentity identity)
{
  Console.WriteLine("Show principal information");
  WindowsPrincipal principal = new WindowsPrincipal(identity);
  if (principal == null)
  {
    Console.WriteLine("not a Windows Principal");
    return null;
  }
  Console.WriteLine($"Users? {principal.IsInRole(WindowsBuiltInRole.User)}");
  Console.WriteLine(
    $"Administrators? {principal.IsInRole(WindowsBuiltInRole.Administrator)}");
  Console.WriteLine();
  return principal;
}
```

运行应用程序，我的账户属于 Users 角色，而不是 Administrator 角色，得到以下结果：

```
Show principal information
Users? True
Administrators? False
```

很明显，如果能很容易地访问当前用户及其角色的详细信息，然后使用那些信息决定允许或拒绝用户执行某些动作，这就非常有好处。利用角色和 Windows 用户组，管理员可以完成使用标准用户管理工具所能完成的工作，这样，在用户的角色改变时，通常可以避免更改代码。

所有 principal 类都派生自基类 ClaimsPrincipal。这样，可以使用 principal 对象的 Claims 属性来访问用户的声称。下一节讨论声称。

### 24.2.3 使用声称

声称(claim)提供了比角色更大的灵活性。声称是一个关于标识(来自权威机构)的语句。权威机构如 Active

Directory 或 Microsoft Live 账户身份验证服务，建立关于用户的声称，例如，用户名的声称、用户所属的组的声称或关于年龄的声称。用户已经 21 岁了，有资格访问特定的资源吗？

方法 ShowClaims 访问一组声称，把主题、发行人、声称类型和更多选项写到控制台(代码文件 WindowsPrincipal/Program.cs)：

```
public static void ShowClaims(IEnumerable<Claim> claims)
{
  Console.WriteLine("Claims");
  foreach (var claim in claims)
  {
    Console.WriteLine($"Subject: {claim.Subject}");
    Console.WriteLine($"Issuer: {claim.Issuer}");
    Console.WriteLine($"Type: {claim.Type}");
    Console.WriteLine($"Value type: {claim.ValueType}");
    Console.WriteLine($"Value: {claim.Value}");
    foreach (var prop in claim.Properties)
    {
      Console.WriteLine($"\tProperty: {prop.Key} {prop.Value}");
    }
    Console.WriteLine();
  }
}
```

下面是从 Microsoft Live 账户中提取的一个声称，它提供了名称、主 ID 和组标识符等信息。

```
Claims
Subject: System.Security.Principal.WindowsIdentity
Issuer: AD AUTHORITY
Type: http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name
Value type: http://www.w3.org/2001/XMLSchema#string
Value: THEROCKS\Christian

Subject: System.Security.Principal.WindowsIdentity
Issuer: AD AUTHORITY
Type: http://schemas.microsoft.com/ws/2008/06/identity/claims/primarysid
Value type: http://www.w3.org/2001/XMLSchema#string
Value: S-1-5-21-1413171511-313453878-1364686672-1001
    Property: http://schemas.microsoft.com/ws/2008/06/identity/claims/
    windowssubauthority NTAuthority

Subject: System.Security.Principal.WindowsIdentity
Issuer: AD AUTHORITY
Type: http://schemas.microsoft.com/ws/2008/06/identity/claims/groupsid
Value type: http://www.w3.org/2001/XMLSchema#string
Value: S-1-1-0
    Property: http://schemas.microsoft.com/ws/2008/06/identity/claims/
    windowssubauthority WorldAuthority

Subject: System.Security.Principal.WindowsIdentity
Issuer: AD AUTHORITY
Type: http://schemas.microsoft.com/ws/2005/05/identity/claims/denyonlysid
Value type: http://www.w3.org/2001/XMLSchema#string
Value: S-1-5-114
    Property: http://schemas.microsoft.com/ws/2008/06/identity/claims/
    windowssubauthority NTAuthority
...
```

可以从声称的提供程序中把声称添加到 Windows 标识。还可以从简单的客户端程序中添加声称，如年龄声称：

```
identity.AddClaim(new Claim("Age", "25"));
```

使用程序中的声称，相信这个声称。这个声称是真的——是 25 岁吗？声称也可以是谎言。从客户机应用程序中添加这个声称，可以看到，声称的发行人是 LOCAL AUTHORITY。AD AUTHORITY (the Active Directory)的信息更值得信赖，但这里需要信任 Active Directory 系统管理员。

WindowsIdentity 派生自基类 ClaimsIdentity，提供了几个方法来检查声称，或检索特定的声称。为了测试声称是否可用，可以使用 HasClaim 方法：

```
bool hasName = identity.HasClaim(c => c.Type == ClaimTypes.Name);
```

要检索特定的声称，FindAll 方法需要一个谓词来定义匹配：

```
var groupClaims = identity.FindAll(c => c.Type == ClaimTypes.GroupSid);
```

**注意：**
*声称类型可以是一个简单的字符串，例如前面使用的"Age"类型。ClaimType 定义了一组已知的类型，例如 Country、Email、Name、MobilePhone、UserData、Surname、PostalCode 等。*

## 24.3 加密数据

机密数据应得到保护，从而使未授权的用户不能读取它们。这对于在网络中发送的数据或存储的数据都有效。可以用对称或不对称密钥来加密这些数据。

通过对称密钥，可以使用同一个密钥进行加密和解密。与不对称的加密相比，加密和解密使用不同的密钥：公钥/私钥。如果使用一个公钥进行加密，就应使用对应的私钥进行解密，而不是使用公钥解密。同样，如果使用一个私钥加密，就应使用对应的公钥解密，而不是使用私钥解密。不可能从私钥中计算出公钥，也不可能从公钥中计算出私钥。

公钥/私钥总是成对创建。公钥可以由任何人使用，它甚至可以放在 Web 站点上，但私钥必须安全地加锁。为了说明加密过程，下面看看使用公钥和私钥的例子。

如果 Alice 给 Bob 发了一封电子邮件，如图 24-1 所示，并且 Alice 希望能保证除了 Bob 外，其他人都不能阅读该邮件，那么她就使用 Bob 的公钥。邮件是使用 Bob 的公钥加密的。Bob 打开该邮件，并使用他秘密存储的私钥解密。这种方式可以保证除了 Bob 外，其他人都不能阅读 Alice 的邮件。

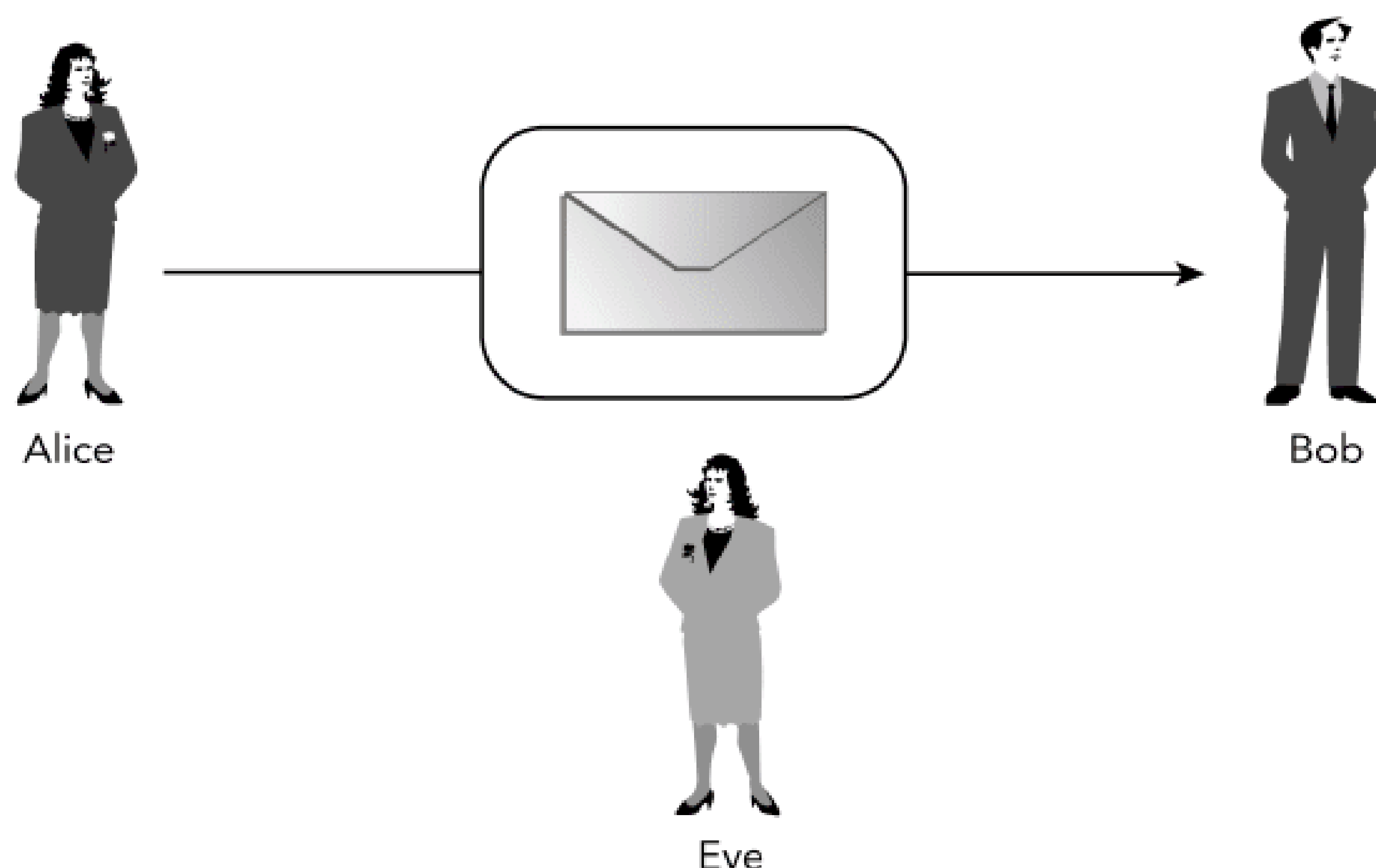


图 24-1

但这还有一个问题：Bob 不能确保邮件是 Alice 发送来的。Eve 可以使用 Bob 的公钥加密发送给 Bob 的邮件并假装是 Alice。我们使用公钥/私钥把这条规则扩展一下。下面再次从 Alice 给 Bob 发送电子邮件开始。在 Alice 使用 Bob 的公钥加密邮件之前，她添加了自己的签名，再使用自己的私钥加密该签名。然后使用 Bob 的公钥加密邮件。这样就保证除 Bob 外，其他人都不能阅读该邮件。在 Bob 解密邮件时，他检测到一个加密的签名。这个签名可以使用 Alice 的公钥来解密。而 Bob 可以访问 Alice 的公钥，因为这个密钥是公钥。在解密了签名后，Bob 就可以确定是 Alice 发送了电子邮件。

使用对称密钥的加密和解密算法比使用非对称密钥的算法快得多。对称密钥的问题是密钥必须以安全的方式互换。在网络通信中，一种方式是先使用非对称的密钥进行密钥互换，再使用对称密钥加密通过网络发送的数据。

.NET在System.Security.Cryptography名称空间中包含用于加密的类。它实现了几个对称算法和非对称算法。有几个不同的算法类用于不同的目的。一些类以Cng作为前缀或后缀。CNG是Cryptography Next Generation的简称，是本机Windows Crypto API的更新版本，这个API可以使用基于提供程序的模型，编写独立于算法的程序。

表 24-1 列出了 System.Security.Cryptography 名称空间中的加密类及其功能。没有 Cng、Managed 或 CryptoServiceProvider 后缀的类是抽象基类，如 MD5。Managed 后缀表示这个算法用托管代码实现，其他类可能封装了本地 Windows API 调用。CryptoServiceProvider 后缀用于实现了抽象基类的类，Cng 后缀用于利用新 Cryptography CNG API 的类。

表 24-1

| 类 别 | 类 | 说 明 |
| --- | --- | --- |
| 散列 | MD5<br>SHA1<br>SHA1Managed<br>SHA256<br>SHA256Managed<br>SHA256Cng<br>SHA384<br>SHA384Managed<br>SHA512<br>SHA512Managed | 散列算法的目标是从任意长度的二进制字符串中创建一个长度固定的散列值。这些算法和数字签名一起用于保证数据的完整性。如果再次散列相同的二进制字符串，会返回相同的散列结果。MD5(Message Digest Algorithm 5，消息摘要算法 5)由 RSA 实验室开发，比 SHA1 快。SHA1 在抵御暴力攻击方面比较强大。SHA 算法由美国国家安全局(NSA)设计。MD5 使用 128 位的散列长度，SHA1 使用 160 位。其他 SHA 算法在其名称中包含了散列长度。SHA512 是这些算法中最强大的，其散列长度为 512 位，它也是最慢的 |
| 对称 | DES DESCryptoServiceProvider<br>TripleDESTripleDESCryptoServiceProvider<br>AesAesCryptoServiceProvider<br>AesManaged RC2<br>RC2CryptoServiceProvider<br>RijandelRijandelManaged | 对称密钥算法使用相同的密钥进行数据的加密和解密。现在认为 DES(Data Encryption Standard，数据加密标准)是不安全的，因为它只使用 56 位的密钥长度，可以在不超过 24 小时的时间内破解。Triple-DES 是 DES 的继承者，其密钥长度是 168 位，但它提供的有效安全性只有 112 位。AES(Advanced Encryption Standard，高级加密标准)的密钥长度是 128、192 或 256 位。Rijandel 非常类似于 AES，它只是在密钥长度方面的选项较多。AES 是美国政府采用的加密标准 |
| 非对称 | DSA<br>DSACryptoServiceProvider<br>ECDsa<br>ECDsaCng<br>ECDiffieHellman<br>ECDiffieHellmanCng<br>RSA<br>RSACryptoServiceProvider<br>RSACng | 非对称算法使用不同的密钥进行加密和解密。RSA(Rivest，Shamir，Adleman)是第一个用于签名和加密的算法。这个算法广泛用于电子商务协议。RSACng 是.NET 4.6 和.NET Core 的一个新类，基于 Cryptography Next Generation (CNG)实现方式。DSA(Digital Signature Algorithm，数字签名算法)是用于数字签名的一个美国联邦政府标准。ECDSA(Elliptic Curve DSA，椭圆曲线数字签名算法)和 EC Diffie-Hellman 使用基于椭圆曲线组的算法。这些算法比较安全，且使用较短的密钥长度。例如，DSA 的密钥长度为 1024 位，其安全性类似于 ECDSA 的 160 位。因此，ECDSA 比较快。EC Diffie-Hellman 算法用于以安全的方式在公共信道中交换私钥 |

下面用例子说明如何通过编程使用这些算法。

**注意：**
带有 Cng 前缀或后缀（属于 Windows Cryptography Next Generation, CNG）的所有类仅在 Windows 上得到支持，不能在 Linux 或 Mac 上运行。这些 API 在 Linux 上会抛出 PlatformNotSupportedException 异常。

### 24.3.1 创建和验证签名

第一个例子说明了如何使用 ECDSA 算法进行签名。Alice 创建了一个签名，它用 Alice 的私钥加密，可以

使用 Alice 的公钥访问。因此保证该签名来自 Alice。

SigningDemo 示例代码使用如下依赖项和名称空间：

**依赖项**

System.Security.Cryptography.Cng

**名称空间**

System

System.Security.Cryptography

System.Text

首先，看看 Main()方法中的主要步骤：创建 Alice 的密钥，给字符串“Alice”签名，最后使用公钥验证该签名是否来自 Alice。要签名的消息使用 Encoding 类转换为一个字节数组。要把加密的签名写入控制台，包含该签名的字节数组应使用 Convert.ToBase64String()方法转换为一个字符串(代码文件 SigningDemo/Program.cs)。

```
private CngKey _aliceKeySignature;
private byte[] _alicePubKeyBlob;

static void Main()
{
  var p = new Program();
  p.Run();
}

public void Run()
{
  InitAliceKeys();
  byte[] aliceData = Encoding.UTF8.GetBytes("Alice");
  byte[] aliceSignature = CreateSignature(aliceData, aliceKeySignature);
  Console.WriteLine($"Alice created signature: " +
    $"{Convert.ToBase64String(aliceSignature)}");
  if (VerifySignature(aliceData, aliceSignature, alicePubKeyBlob))
  {
    Console.WriteLine("Alice signature verified successfully");
  }
}
```

**注意：**

千万不要使用 Encoding 类把加密的数据转换为字符串。Encoding 类验证和转换 Unicode 不允许使用的无效值，因此把字符串转换回字节数组会得到另一个结果。

InitAliceKeys()方法为 Alice 创建新的密钥对。因为这个密钥对存储在一个静态字段中，所以可以从其他方法中访问它。CngKey 类的 Create()方法把该算法作为一个参数，为算法定义密钥对。通过 Export()方法，导出密钥对中的公钥。这个公钥可以提供给 Bob，来验证签名。Alice 保留其私钥。除了使用 CngKey 类创建密钥对之外，还可以打开存储在密钥存储器中的已有密钥。通常 Alice 在其私有存储器中有一个证书，其中包含了一个密钥对，该存储器可以用 CngKey.Open()方法访问。

```
private void InitAliceKeys()
{
  _aliceKeySignature = CngKey.Create(CngAlgorithm.ECDsaP521);
  _alicePubKeyBlob =
    aliceKeySignature.Export(CngKeyBlobFormat.GenericPublicBlob);
}
```

有了密钥对，Alice 就可以使用 ECDsaCng 类创建签名了。这个类的构造函数从 Alice 那里接收包含公钥和私钥的 CngKey 类。再使用私钥，通过 SignData()方法给数据签名：

```
public byte[] CreateSignature(byte[] data, CngKey key)
{
  byte[] signature;
  using (var signingAlg = new ECDsaCng(key))
  {
    signature = signingAlg.SignData(data, HashAlgorithmName.SHA512);
    signingAlg.Clear();
  }
  return signature;
}
```

要验证签名是否来自于 Alice，Bob 使用 Alice 的公钥检查签名。包含公钥 blob 的字节数组可以用静态方法 Import()导入 CngKey 对象。然后使用 ECDsaCng 类，调用 VerifyData()方法来验证签名。

```
public bool VerifySignature(byte[] data, byte[] signature, byte[] pubKey)
{
  bool retValue = false;
  using (CngKey key = CngKey.Import(pubKey, CngKeyBlobFormat.GenericPublicBlob))
  using (var signingAlg = new ECDsaCng(key))
  {
    retValue = signingAlg.VerifyData(data, signature, HashAlgorithmName.SHA512);
    signingAlg.Clear();
  }
  return retValue;
}
```

## 24.3.2　实现安全的数据交换

下一个例子帮助解释公钥/私钥的原则，在两个团体之间交换机密数据，用对称密钥通信。它使用 EC Diffie-Hellman 算法在两个团体之间交换机密数据。这个算法允许仅使用公钥和私钥来交换机密数据，在两个团体之间交换公钥。编写本书时，这个算法的实现代码还不能用于.NET Core，只能用于运行在 Windows 上的.NET Framework。在应用程序中，一般不需要实现这个功能，因为基础架构服务已经提供了它。

**注意：**

编写本书时，.NET Core 仅包含 ECDiffieHellman 抽象基类，实现代码可以使用它创建具体的类。目前还没有具体的类，所以这个示例仅使用.NET 4.7.1。

SecureTransfer 示例应用程序的目标框架设置为 net471，使用如下依赖项和名称空间：

**依赖项**

System.Security.Cryptography.Algorithms

System.Security.Cryptography.Cng

System.Security.Cryptography.Csp

System.Security.Cryptography.Primitives

**名称空间**

System

System.IO

System.Security.Cryptography

System.Text

System.Threading.Tasks

Main()方法包含了其主要功能。Alice 创建了一条加密的消息，并把它发送给 Bob。在此之前，要先为 Alice 和 Bob 创建密钥对。Bob 只能访问 Alice 的公钥，Alice 也只能访问 Bob 的公钥(代码文件 SecureTransfer/Program.cs)。

```
private CngKey _aliceKey;
private CngKey _bobKey;
private byte[] _alicePubKeyBlob;
private byte[] _bobPubKeyBlob;

static async Task Main()
{
  var p = new Program();
  await p.RunAsync();
  Console.ReadLine();
}

public async Task RunAsync()
{
  try
  {
    CreateKeys();
    byte[] encrytpedData =
```

```
      await AliceSendsDataAsync("This is a secret message for Bob");
    await BobReceivesDataAsync(encrytpedData);
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

在 CreateKeys()方法的实现代码中，使用 EC Diffie-Hellman 512 算法创建密钥。

```
public void CreateKeys()
{
  aliceKey = CngKey.Create(CngAlgorithm.ECDiffieHellmanP521);
  bobKey = CngKey.Create(CngAlgorithm.ECDiffieHellmanP521);
  alicePubKeyBlob = aliceKey.Export(CngKeyBlobFormat.EccPublicBlob);
  bobPubKeyBlob = bobKey.Export(CngKeyBlobFormat.EccPublicBlob);
}
```

在 AliceSendsDataAsync()方法中，包含文本字符的字符串使用 Encoding 类转换为一个字节数组。创建一个 ECDiffieHellmanCng 对象，用 Alice 的密钥对初始化它。Alice 调用 DeriveKeyMaterial()方法，从而使用其密钥对和 Bob 的公钥创建一个对称密钥。返回的对称密钥使用对称算法 AES 加密数据。AesCryptoServiceProvider 需要密钥和一个初始化矢量(IV)。IV 从 GenerateIV()方法中动态生成，对称密钥用 EC Diffie-Hellman 算法交换，但还必须交换 IV。从安全性角度来看，在网络上传输未加密的 IV 是可行的——只是密钥交换必须是安全的。IV 存储为内存流中的第一项内容，其后是加密的数据，其中，CryptoStream 类使用 AesCryptoServiceProvider 类创建的 encryptor。在访问内存流中的加密数据之前，必须关闭加密流。否则，加密数据就会丢失最后的位。

```
public async Task<byte[]> AliceSendsDataAsync(string message)
{
  Console.WriteLine($"Alice sends message: {message}");
  byte[] rawData = Encoding.UTF8.GetBytes(message);
  byte[] encryptedData = null;
  using (var aliceAlgorithm = new ECDiffieHellmanCng(aliceKey))
  using (CngKey bobPubKey = CngKey.Import(bobPubKeyBlob,
    CngKeyBlobFormat.EccPublicBlob))
  {
    byte[] symmKey = aliceAlgorithm.DeriveKeyMaterial(bobPubKey);
    Console.WriteLine("Alice creates this symmetric key with " +
      $"Bobs public key information: {Convert.ToBase64String(symmKey)}");
    using (var aes = new AesCryptoServiceProvider())
    {
      aes.Key = symmKey;
      aes.GenerateIV();
      using (ICryptoTransform encryptor = aes.CreateEncryptor())
      using (var ms = new MemoryStream())
      {
        // create CryptoStream and encrypt data to send
        using (var cs = new CryptoStream(ms, encryptor,
        CryptoStreamMode.Write))
        {
          // write initialization vector not encrypted
          await ms.WriteAsync(aes.IV, 0, aes.IV.Length);
          cs.Write(rawData, 0, rawData.Length);
        }
        encryptedData = ms.ToArray();
      }
      aes.Clear();
    }
  }
  Console.WriteLine("Alice: message is encrypted: "+
    $"{Convert.ToBase64String(encryptedData)}");
  Console.WriteLine();
  return encryptedData;
}
```

Bob 从 BobReceivesDataAsync()方法的参数中接收加密数据。首先，必须读取未加密的初始化矢量。AesCryptoServiceProvider 类的 BlockSize 属性返回块的位数。位数除以 8，就可以计算出字节数。最快的方式是把数据右移 3 位。右移 1 位就是除以 2，右移 2 位就是除以 4，右移 3 位就是除以 8。在 for 循环中，包含未加密 IV 的原字节的前几个字节写入数组 iv 中。接着用 Bob 的密钥对实例化一个 ECDiffieHellmanCng 对象。使用 Alice 的公钥，从 DeriveKeyMaterial()方法返回对称密钥。

比较 Alice 和 Bob 创建的对称密钥，可以看出所创建的密钥值相同。使用这个对称密钥和初始化矢量，来自 Alice 的消息就可以用 AesCryptoServiceProvider 类解密。

```
public async Task BobReceivesDataAsync(byte[] encryptedData)
{
  Console.WriteLine("Bob receives encrypted data");
  byte[] rawData = null;
  var aes = new AesCryptoServiceProvider();
  int nBytes = aes.BlockSize 3;
  byte[] iv = new byte[nBytes];
  for (int i = 0; i < iv.Length; i++)
  {
    iv[i] = encryptedData[i];
  }
  using (var bobAlgorithm = new ECDiffieHellmanCng(bobKey))
  using (CngKey alicePubKey = CngKey.Import(alicePubKeyBlob,
    CngKeyBlobFormat.EccPublicBlob))
  {
    byte[] symmKey = bobAlgorithm.DeriveKeyMaterial(alicePubKey);
    Console.WriteLine("Bob creates this symmetric key with " +
      $"Alices public key information: {Convert.ToBase64String(symmKey)}");
    aes.Key = symmKey;
    aes.IV = iv;
    using (ICryptoTransform decryptor = aes.CreateDecryptor())
    using (MemoryStream ms = new MemoryStream())
    {
      using (var cs = new CryptoStream(ms, decryptor, CryptoStreamMode.Write))
      {
        await cs.WriteAsync(encryptedData, nBytes,
          encryptedData.Length - nBytes);
      }
      rawData = ms.ToArray();
      Console.WriteLine("Bob decrypts message to: " +
        $"{Encoding.UTF8.GetString(rawData)}");
    }
    aes.Clear();
  }
}
```

运行应用程序，会在控制台上看到如下输出。来自 Alice 的消息被加密，Bob 用安全交换的对称密钥解密。

```
Alice sends message: this is a secret message for Bob
Alice creates this symmetric key with Bobs public key information:
q4D182m7lyev9Nlp6f0av2Jvc0+LmHF5zEjXw1O1I3Y=
Alice: message is encrypted: WpOxvUoWH5XY31wC8aXcDWeDUWa6zaSObfGcQCpKixzlTJ9exb
tkF5Hp2WPSZWL9V9n13toBg7hgjPbrVzN2A==
Bob receives encrypted data
Bob creates this symmetric key with Alices public key information:
q4D182m7lyev9Nlp6f0av2Jvc0+LmHF5zEjXw1O1I3Y=
Bob decrypts message to: this is a secret message for Bob
```

### 24.3.3 使用 RSA 签名和散列

一个新的加密算法类是 RSACng。RSA(这个名字来自于算法设计者 Ron Rivest、Adi Shamir 和 Leonard Adlerman)是一个广泛使用的非对称算法。RSA 算法已经可用于.NET、RSA 和 RSACryptoServiceProvider 类，RSACng 类基于 CNG API，其用法类似于先前使用的 ECDSACng 类。

对于本节所示的示例应用程序，Alice 创建一个文档，散列它，以确保它不会改变，给它加上签名，保证是 Alice 生成了文档。Bob 接收文件，并检查 Alice 的担保，以确保文件没有被篡改。

RSA 示例代码使用了如下依赖项和名称空间：

**依赖项**

System.Security.Cryptography.Cng

**名称空间**

System

System.IO

System.Linq

构造应用程序的 Main()方法，开始 Alice 的任务，调用方法 AliceTasks()，来创建一个文档、散列码和签名。然后把这些信息传递给 Bob 的任务，调用方法 BobTasks()(代码文件 RSASample/Program.cs)：

```
class Program
{
  private CngKey _aliceKey;
  private byte[] _alicePubKeyBlob;

  static void Main()
  {
    var p = new Program();
    p.Run();
  }

  public void Run()
  {
    AliceTasks(out byte[] document, out byte[] hash, out byte[] signature);
    BobTasks(document, hash, signature);
  }
  //...
}
```

方法 AliceTasks()首先创建 Alice 所需的密钥，将消息转换为一个字节数组，散列字节数组，并添加一个签名：

```
public void AliceTasks(out byte[] data, out byte[] hash, out byte[] signature)
{
  InitAliceKeys();
  data = Encoding.UTF8.GetBytes("Best greetings from Alice");
  hash = HashDocument(data);
  signature = AddSignatureToHash(hash, _aliceKey);
}
```

与之前一样，Alice 所需的密钥是使用 CngKey 类创建的。现在正在使用 RSA 算法，把 CngAlgorithm.Rsa 枚举值传递到 Create 方法，来创建公钥和私钥。公钥只提供给 Bob，所以公钥用 Export 方法提取：

```
private void InitAliceKeys()
{
  _aliceKey = CngKey.Create(CngAlgorithm.Rsa);
  _alicePubKeyBlob = _aliceKey.Export(CngKeyBlobFormat.GenericPublicBlob);
}
```

从 Alice 的任务中调用 HashDocument 方法，为文档创建一个散列码。散列码使用一个散列算法 SHA384 类创建。不管文档有多长，散列码的长度总是相同。再次为相同的文档创建散列码，会得到相同的散列码。Bob 需要在文档上使用相同的算法。如果返回相同的散列码，就说明文档没有改变。

```
private byte[] HashDocument(byte[] data)
{
  using (var hashAlg = SHA384.Create())
  {
    return hashAlg.ComputeHash(data);
  }
}
```

添加签名，可以保证文档来自 Alice。在这里，使用 RSACng 类给散列签名。Alice 的 CngKey(包括公钥和私钥)传递给 RSACng 类的构造函数；签名通过调用 SignHash 方法创建。给散列签名时，SignHash 方法需要了解散列算法；HashAlgorithmName.SHA384 是创建散列所使用的算法。此外，需要 RSA 填充。RSASignaturePadding 枚举的可能选项是 Pss 和 Pkcs1：

```
private byte[] AddSignatureToHash(byte[] hash, CngKey key)
{
  using (var signingAlg = new RSACng(key))
  {
    byte[] signed = signingAlg.SignHash(hash,
      HashAlgorithmName.SHA384, RSASignaturePadding.Pss);
    return signed;
  }
}
```

Alice 散列并签名后，Bob 的任务可以在 BobTasks 方法中开始。Bob 接收文档数据、散列码和签名，他使用 Alice 的公钥。首先，Alice 的公钥使用 CngKey. Import 导入，分配给 aliceKey 变量。接下来，Bob 使用辅助

方法 IsSignatureValid 和 IsDocumentUnchanged，来验证签名是否有效，文档是否不变。只有在两个条件是 true 时，文档才写入控制台：

```
public void BobTasks(byte[] data, byte[] hash, byte[] signature)
{
  CngKey aliceKey = CngKey.Import(_alicePubKeyBlob,
  CngKeyBlobFormat.GenericPublicBlob);
  if (!IsSignatureValid(hash, signature, aliceKey))
  {
    Console.WriteLine("signature not valid");
    return;
  }
  if (!IsDocumentUnchanged(hash, data))
  {
    Console.WriteLine("document was changed");
    return;
  }
  Console.WriteLine("signature valid, document unchanged");
  Console.WriteLine($"document from Alice: {Encoding.UTF8.GetString(data)}");
}
```

为了验证签名是否有效，使用 Alice 的公钥创建 RSACng 类的一个实例。通过这个类，使用 VerifyHash 方法传递散列、签名、早些时候使用的算法信息。现在 Bob 知道，信息来自 Alice：

```
private bool IsSignatureValid(byte[] hash, byte[] signature, CngKey key)
{
  using (var signingAlg = new RSACng(key))
  {
    return signingAlg.VerifyHash(hash, signature, HashAlgorithmName.SHA384,
      RSASignaturePadding.Pss);
  }
}
```

为了验证文档数据没有改变，Bob 再次散列文件，并使用 LINQ 扩展方法 SequenceEqual，验证散列码是否与早些时候发送的相同。如果散列值是相同的，就可以假定文档没有改变：

```
private bool IsDocumentUnchanged(byte[] hash, byte[] data)
{
  byte[] newHash = HashDocument(data);
  return newHash.SequenceEqual(hash);
}
```

运行应用程序，输出如下。调试应用程序时，可以在 Alice 散列后修改文档数据，Bob 不会接受更改的文档。为了改变文档数据，很容易在调试器的 Watch 窗口中改变值。

```
signature valid, document unchanged
document from Alice: Best greetings from Alice
```

## 24.4 保护数据

在加密、签名和散列数据之后，下面将抽象层移动到更高的位置。本节将讨论数据的保护——存储对安全性敏感的数据。当数据与 Web 应用程序一起使用时，所使用的客户端是不可信任的。这就需要使用数据保护 API 了。

另一种需要保护的数据是配置数据。配置数据——比如 SQL 连接字符串(包括用户名和密码)或 AWS 或微软 Azure 的访问令牌——不应该放在公共源代码存储库中。这些信息很容易被误用。实际上，机器人被用来在 GitHub 的公共存储库中爬行，寻找密钥并使用它们，例如，运转虚拟机来存放比特币。App Secrets 提供了一种方法，可以将机密数据存储在用户的配置文件中。

这些技术、数据保护和用户机密都在本节中讨论。

### 24.4.1 实现数据保护

在.NET Framework 中，名称空间 System.Security. DataProtection 包含 DpApiDataProtector 类，而这个类包装了本机 Windows Data Protection API (DPAPI)。这些类基于 Windows，并不提供.NET Core 需要的灵活性和功能，

所以 ASP.NET 团队创建了 Microsoft.AspNetCore.DataProtection 名称空间中的类。

使用这个库的原因是为日后的检索存储可信的信息，但存储媒体(如使用第三方的托管环境)不能信任自己，所以信息需要加密存储在主机上。

示例应用程序是一个简单的控制台应用程序(.NET Core)，允许使用数据保护功能读写信息。在这个示例中，可以看到 ASP.NET 数据保护的灵活性和功能。

数据保护的示例代码使用了以下依赖项和名称空间：

**依赖项**

Microsoft.AspNetCore.DataProtection

Microsoft.Extensions.DependencyInjection

**名称空间**

Microsoft.AspNetCore.DataProtection

Microsoft.Extensions.DependencyInjection

System

System.IO

System.Linq

使用-r 和-w 命令行参数，可以启动控制台应用程序，读写存储器。此外，需要使用命令行，设置一个文件名来读写。检查命令行参数后，通过调用 SetupDataProtection 辅助方法来初始化数据保护。这个方法返回一个 MySafe 类型的对象，嵌入 IDataProtector。之后，根据命令行参数，调用 Write 或 Read 方法(代码文件 DataProtectionSample/Program.cs)：

```
class Program
{
  private const string readOption = "-r";
  private const string writeOption = "-w";
  private readonly string[] options = { readOption, writeOption };
  static void Main(string[] args)
  {
    if (args.Length != 2 || args.Intersect(options).Count() != 1)
    {
      ShowUsage();
      return;
    }

    string fileName = args[1];
    MySafe safe = SetupDataProtection();
    switch (args[0])
    {
      case writeOption:
        Write(safe, fileName);
        break;
      case readOption:
        Read(safe, fileName);
        break;
      default:
        ShowUsage();
        break;
    }
  }
  //...
}
```

类 MySafe 有一个 IDataProtector 成员。这个接口定义了成员 Protect 和 Unprotect，来加密和解密数据。这个接口定义了 Protect 和 Unprotect 方法，这些方法带有字节数组参数，返回字节数组。不过，示例代码使用 NuGet 包 Microsoft.AspNetCore.DataProtection.Abstractions 中定义的扩展方法，直接发送、返回来自 Encrypt 和 Decrypt 方法的字符串。MySafe 类通过依赖注入接收 IDataProtectionProvider 接口。有了这个接口，传递目的字符串，返回 IDataProtector。读写这个安全时，需要使用相同的字符串 (代码文件 DataProtectionSample/MySafe.cs)：

```
public class MySafe
{
  private IDataProtector _protector;
  public MySafe(IDataProtectionProvider provider) =>
```

```
    _protector = provider.CreateProtector("MySafe.MyProtection.v2");

  public string Encrypt(string input) => _protector.Protect(input);
  public string Decrypt(string encrypted) => _protector.Unprotect(encrypted);
}
```

在 SetupDataProtection 方法中，调用 AddDataProtection 扩展方法，通过依赖注入添加数据保护，并配置它。AddDataProtection 方法注册默认服务，返回一个 AddDataProtection，它可进一步使用流利的 API，配置数据保护。示例代码把 DirectoryInfo 实例传递给 PersistKeysToFileSystem 方法，把密钥保存在实际的目录中。另一个选择是把密钥保存到注册表(PersistKeysToRegistry)中，可以创建自己的方法，把密钥保存在定制的存储中。所创建密钥的生命周期由 SetDefaultKeyLifetime 方法定义。接下来，密钥通过调用 ProtectKeysWithDpapi 来保护。这个方法使用 DPAPI 保护密钥，加密与当前用户一起存储的密钥。ProtectKeysWithCertificate 允许使用证书保护密钥。API 还定义了 UseEphemeralDataProtectionProvider 方法，把密钥存储在内存中。再次启动应用程序时，需要生成新密钥。这个功能非常适合于单元测试(代码文件 DataProtectionSample/Program.cs)：

```
public static MySafe SetupDataProtection()
{
  var serviceCollection = new ServiceCollection();
  serviceCollection.AddDataProtection()
    .PersistKeysToFileSystem(new DirectoryInfo("."))
    .SetDefaultKeyLifetime(TimeSpan.FromDays(20))
    .ProtectKeysWithDpapi();

  IServiceProvider services = serviceCollection.BuildServiceProvider();

  return ActivatorUtilities.CreateInstance<MySafe>(services);
}
```

**注意：**
依赖注入参见第 20 章。

现在，实现了数据保护应用程序的核心，Write 和 Read 方法可以利用 MySafe，加密和解密用户的内容：

```
public static void Write(MySafe safe, string fileName)
{
  Console.WriteLine("enter content to write:");
  string content = Console.ReadLine();
  string encrypted = safe.Encrypt(content);
  File.WriteAllText(fileName, encrypted);
  Console.WriteLine($"content written to {fileName}");
}

public static void Read(MySafe safe, string fileName)
{
  string encrypted = File.ReadAllText(fileName);
  string decrypted = safe.Decrypt(encrypted);
  Console.WriteLine(decrypted);
}
```

### 24.4.2　用户机密

只要使用 Windows 身份验证，连接字符串放在配置文件中就不是大问题。使用连接字符串存储用户名和密码时，将连接字符串添加到配置文件，并将配置文件与源代码存储库一起存储可能是一个大问题。拥有一个公共存储库，并使用配置存储 Amazon 或 Azure 密钥，可能会很快导致损失数千美元。黑客的后台工作在公开的 GitHub 库中搜索，寻找 Amazon 密钥来劫持账户，并创建制造比特币的虚拟机。访问 https://www.humankode.com/security/how-a-bug-in-visual-studio-2015-exposed-my-source-code-on-github-and-cost-me-6500-in-a-few-hours 来了解更多情况。

.NET Core 在这一点上有一些缓和：用户机密。有了用户机密，配置就不会存储在项目的配置文件中，它存储在与账户相关联的配置文件中。

用户机密示例代码使用了以下依赖项和名称空间：

**依赖项**

Microsoft.Extensions.Configuration

Microsoft.Extensions.Configuration.CommandLine
Microsoft.Extensions.Configuration.EnvironmentVariables
Microsoft.Extensions.Configuration.Json
Microsoft.Extensions.Configuration.UserSecrets

**名称空间**

Microsoft.Extensions.Configuration
System
System.IO

要获得 dotnet CLI 工具的用户机密命令行扩展，需要给 csproj 文件添加对 Microsoft.Extensions.SecretManager.Tools 的引用(项目文件 UserSecretsSample/UserSecretsSample.csproj)：

```
<ItemGroup>
  <DotNetCliToolReference Include="Microsoft.Extensions.SecretManager.Tools"
    Version="2.0.0" />
</ItemGroup>
```

此外，需要在项目文件中向 UserSecretsId 元素添加一个值，来定义初始 ID(项目文件 UserSecretsSample/UserSecretsSample.csproj)：

```
<PropertyGroup>
  <OutputType>Exe</OutputType>
  <TargetFramework>netcoreapp2.0</TargetFramework>
  <UserSecretsId>UserSecretsSample-Id</UserSecretsId>
</PropertyGroup>
```

这个机密 ID 只需要在本地系统上是唯一的，以避免混淆来自不同项目的配置值。因此，最好将项目名称包含在标识符中。

示例应用程序使用 JSON 文件中的配置、环境变量、命令行和用户机密进行定义。用户机密配置为使用 ConfigurationBuilder 调用扩展方法 AddUserSecrets。只有在启用了调试模式的情况下构建应用程序，才添加此提供程序。请记住，只有使用具有用户机密的用户配置文件运行应用程序，该提供程序才可用。AddUserSecrets 的重载方法需要把用户机密 ID 作为参数——与配置使用的标识符相同(代码文件 UserSecretsSample/Program.cs)：

```
static void Main(string[] args)
{
  var configBuilder = new ConfigurationBuilder();
  configBuilder.SetBasePath(Directory.GetCurrentDirectory())
    .AddJsonFile("appsettings.json")
    .AddEnvironmentVariables()
    .AddCommandLine(args);

#if DEBUG
  configBuilder.AddUserSecrets("UserSecretsSample-Id");
#endif
  IConfigurationRoot configuration = configBuilder.Build();
  //...
}
```

**注意：**
第 30 章介绍了使用 Microsoft.Extensions.Configuration 进行应用程序配置的更多内容。

使用 JSON 配置文件 appsettings.json，写入键 NotASecret 的值。当项目添加到公共源代码存储库中时，该配置文件不应该包含敏感的配置信息(配置文件 UserSecretsSample/appsettings.json)：

```
{
  "NotASecret": "this is not a secret"
}
```

因为 SecretManager 工具已经配置了项目文件，所以可以使用 dotnet 命令行工具来设置、列出、删除和清除机密。下面的命令使用密钥 Secret1 设置机密：

```
> dotnet user-secrets set Secret1 "this is a secret"
```

使用 dotnet user-secrets list 显示所有用户的机密，使用 dotnet user-secrets clear 可以清除所有用户的机密。

在 Windows 系统上，用户机密存储在文件夹%APPDATA%\microsoft\UserSecrets\ <userSecretsId>\secrets.json 中。在 Linux 系统上，用户机密存储在~/.microsoft/~/usersecrets/<userSecretsId>/secrets.json 中。

要读取配置值，无论配置存储在何处，都可以使用 IConfigurationRoot 类型的变量。使用索引器，可以读取使用相应键存储的值，无论配置是存储在 JSON 文件中还是带有用户机密(代码文件 UserSecretsSample/Program.cs)：

```
string notASecret1 = configuration["NotASecret"];
Console.WriteLine($"not a secret: {notASecret1}");

string secretValue1 = configuration["Secret1"];
Console.WriteLine($"secret: {secretValue1}");
```

在生产系统上，存储用户机密的私有配置文件不可用。根据所使用的技术，可以使用不同的提供程序。例如，通过 Azure App Services，可以使用 Azure 门户指定配置，并使用环境变量检索配置。还可以使用 Azure Key Vault 配置提供程序从 Azure Key Vault 机密中读取配置。

## 24.5　资源的访问控制

在操作系统中，资源(如文件和注册表键，以及命名管道的句柄)都使用访问控制列表(ACL)来保护。图 24-2 显示了这个映射的结构。资源有一个关联的安全描述符。安全描述符包含了资源拥有者的信息，并引用了两个访问控制列表：自由访问控制列表(Discretionary Access Control List，DACL)和系统访问控制列表(System Access Control List，SACL)。DACL 用来确定谁有访问权；SACL 用来确定安全事件日志的审核规则。ACL 包含一个访问控制项(Access Control Entries，ACE)列表。ACE 包含类型、安全标识符和权限。在 DACL 中，ACE 的类型可以是允许访问或拒绝访问。可以用文件设置和获得的权限是创建、读取、写入、删除、修改、改变许可和获得拥有权。

读取和修改访问控制的类在 System.Security.AccessControl 名称空间中。下面的程序说明了如何从文件中读取访问控制列表。

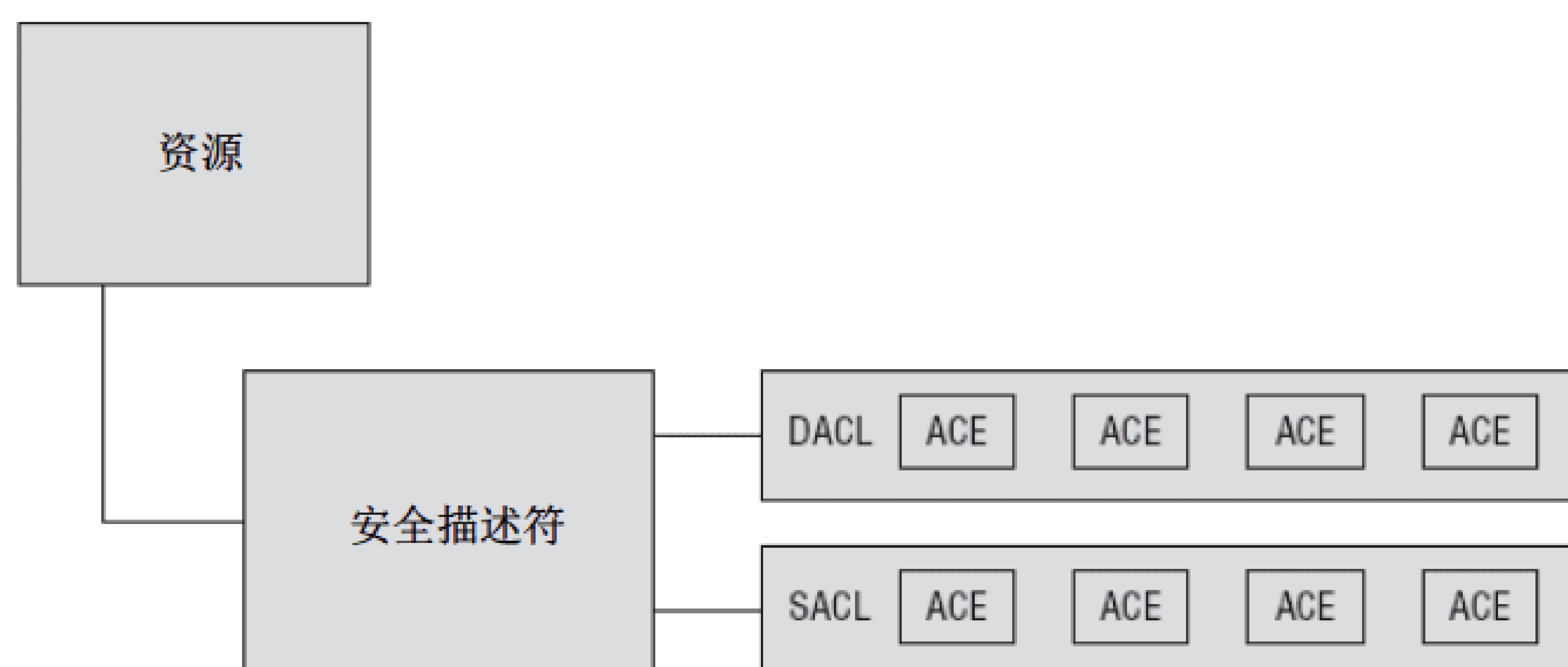



图 24-2

FileAccessControl 示例应用程序使用了如下依赖项和名称空间：

**依赖项**

System.IO.FileSystem

System.IO.FileSystem.AccessControl

**名称空间**

System

System.IO

System.Security.AccessControl

System.Security.Principal

**警告：**
访问控制 API 只能用于 Windows，不能用于 Linux 或 Mac。System.IO.FileSystem.AccessControl *API* 是 Windows 上资源管理的一部分。

FileStream类定义了GetAccessControl()方法，该方法返回一个FileSecurity对象。FileSecurity是一个.NET类，它表示文件的安全描述符。FileSecurity类派生自基类ObjectSecurity、CommonObjectSecurity、NativeObjectSecurity和FileSystemSecurity。其他表示安全描述符的类有CryptoKeySecurity、EventWaitHandleSecurity、MutexSecurity、RegistrySecurity、SemaphoreSecurity、PipeSecurity和ActiveDirectorySecurity。所有这些对象都可以使用访问控制列表来保护。一般情况下，对应的.NET类定义了GetAccessControl()方法，返回相应的安全类；例如，Mutex.GetAccessControl() 方法返回一个 MutexSecurity 类， PipeStream.GetAccessControl() 方法返回一个 PipeSecurity类。

FileSecurity 类定义了读取、修改 DACL 和 SACL 的方法。GetAccessRules()方法以 AuthorizationRuleCollection 类的形式返回 DACL。要访问 SACL，可以使用 GetAuditRules 方法。

在 GetAccessRules()方法中，可以确定是否应使用继承的访问规则(不仅仅是用对象直接定义的访问规则)。最后一个参数定义了应返回的安全标识符的类型。这个类型必须派生自基类 IdentityReference。可能的类型有 NTAccount 和 SecurityIdentifier。这两个类都表示用户或组。NTAccount 类按名称查找安全对象，SecurityIdentifier 类按唯一的安全标识符查找安全对象。

返回的 AuthorizationRuleCollection 包含 AuthorizationRule 对象。AuthorizationRule 对象是 ACE 的.NET 表示。在这里的例子中，因为访问一个文件，所以 AuthorizationRule 对象可以强制转换为 FileSystemAccessRule 类型。在其他资源的 ACE 中，存在不同的.NET 表示，例如 MutexAccessRule 和 PipeAccessRule。在 FileSystemAccessRule 类中，AccessControlType、FileSystemRights 和 IdentityReference 属性返回 ACE 的相关信息(代码文件 FileAccessControl/Program.cs)。

```
class Program
{
  static void Main(string[] args)
  {
    string filename = null;
    if (args.Length == 0) return;
    filename = args[0];
    using (FileStream stream = File.Open(filename, FileMode.Open))
    {
      FileSecurity securityDescriptor = stream.GetAccessControl();
      AuthorizationRuleCollection rules =
        securityDescriptor.GetAccessRules(true, true,
          typeof(NTAccount));
      foreach (AuthorizationRule rule in rules)
      {
        var fileRule = rule as FileSystemAccessRule;
        Console.WriteLine($"Access type: {fileRule.AccessControlType}");
        Console.WriteLine($"Rights: {fileRule.FileSystemRights}");
        Console.WriteLine($"Identity: {fileRule.IdentityReference.Value}");
        Console.WriteLine();
      }
    }
  }
}
```

运行应用程序，并传递一个文件名，就可以看到文件的访问控制列表。这里的输出列出了管理员和系统的全部控制权限、通过身份验证的用户的修改权限，以及属于 Users 组的所有用户的读取和执行权限。

```
Access type: Allow
Rights: FullControl
Identity: BUILTIN\Administrators
Access type: Allow
Rights: FullControl
Identity: NT AUTHORITY\SYSTEM
Access type: Allow
Rights: FullControl
Identity: BUILTIN\Administrators
Access type: Allow
```

```
Rights: FullControl
Identity: TheOtherSide\Christian
```

设置访问权限非常类似于读取访问权限。要设置访问权限，几个可以得到保护的资源类提供了 SetAccessControl()和 ModifyAccessControl()方法。这里的示例代码调用 File 类的 SetAccessControl()方法，以修改文件的访问控制列表。给这个方法传递一个 FileSecurity 对象。FileSecurity 对象用 FileSystemAccessRule 对象填充。这里列出的访问规则拒绝 Sales 组的写入访问权限，给 Everyone 组提供了读取访问权限，并给 Developers 组提供了全部控制权限。

**注意：**
只有定义了 Windows 组 Sales 和 Developers，这个程序才能在系统上运行。可以修改程序，使用自己环境下的可用组。

```
private void WriteAcl(string filename)
{
  var salesIdentity = new NTAccount("Sales");
  var developersIdentity = new NTAccount("Developers");
  var everyOneIdentity = new NTAccount("Everyone");
  var salesAce = new FileSystemAccessRule(salesIdentity,
  FileSystemRights.Write, AccessControlType.Deny);
  var everyoneAce = new FileSystemAccessRule(everyOneIdentity,
  FileSystemRights.Read, AccessControlType.Allow);
  var developersAce = new FileSystemAccessRule(developersIdentity,
  FileSystemRights.FullControl, AccessControlType.Allow);
  var securityDescriptor = new FileSecurity();
  securityDescriptor.SetAccessRule(everyoneAce);
  securityDescriptor.SetAccessRule(developersAce);
  securityDescriptor.SetAccessRule(salesAce);
  File.SetAccessControl(filename, securityDescriptor);
}
```

**注意：**
打开 Properties 窗口，在 Windows 资源管理器中选择一个文件，选择 Security 选项卡，列出访问控制列表，就可以验证访问规则。

# 24.6 Web 安全性

对于允许用户输入的应用程序，Web 应用程序有一些需要注意的特定安全问题。用户输入是不能信任的。使用 JavaScript 或内置 HTML5 特性在客户端验证输入数据只是为了方便用户。可以显示错误，而不向服务器发出额外的网络请求。但是，客户端是不能信任的。因为用户(黑客)可以拦截 HTTP 请求，发出不同的请求，绕过 HTML5 和 JavaScript 验证。

本节讨论 Web 应用程序的常见问题，以及需要注意的避免这些问题的内容。

**注意：**
本节使用的是 ASP.NET Core 和 ASP.NET MVC。更多关于这些技术的信息，请阅读第 30 章。

## 24.6.1 编码

千万不要相信用户输入。将用户信息写入数据库，并将这些信息显示在网站上，是黑客攻击的典型原因。例如，社区网站在主页上显示了最近的 5 个新用户。其中一个新用户设法向用户名添加了一个脚本，该脚本重定向到一个恶意网站上。因为用户信息显示给每个访问该站点的用户，所以每个用户都被重定向。

下一个例子说明了模拟和避免这种行为是多么容易。在下面的代码片段中，/echo URL 映射到一个应答，该应答返回用户输入，并分配给 x 参数，然后使用 context.Response.WriteAsync 发送响应(代码文件 ASPNETCoreMVCSecurity/Startup.cs)：

```
app.Map("/echo", app1 =>
```

```
{
  app1.Run(async context =>
  {
    string data = context.Request.Query["x"];
    await context.Response.WriteAsync(data);
  });
});
```

现在传递请求:

```
http://localhost:24897/echo?x=I'm a nice user
```

字符串 I'm a nice user 返回到浏览器。这并不是那么糟糕，但是用户的运行结果可能不一样。例如，用户可以输入如下 HTML 代码:

```
http://localhost:24897/echo?x=<h1>Is this wanted?</h1>
```

结果显示，输入字符串用 HTML H1 标记格式化。用户可以通过输入 JavaScript 代码做更多的坏事:

```
localhost:24897/echo?x=<script>alert("this is bad");</script>
```

在大多数浏览器中，会出现一个弹出窗口，用户在其中输入文本。如果用 Google Chrome 来尝试，可以用一个单独的措施来避免这个问题。该浏览器显示了一个错误页面 This page isn't working，和附加信息 ERR_BLOCKED_BY_XSS_AUDITOR。

如果试图检查带有输入的<Script >元素，而在这种情况下没有返回值来避免这个问题，就可能会失败。除了使用<Script >元素之外，也可以给尖括号使用 Unicode 数字来得到相同的结果。下面对用户输入进行编码，使其不被浏览器解释。

可以从名称空间 System.Text.Encodings.Web 使用 HTMLEncoder 类来编码用户输入:

```
app.Map("/echoenc", app1 =>
{
  app1.Run(async context =>
  {
    string data = context.Request.Query["x"];
    await context.Response.WriteAsync(HtmlEncoder.Default.Encode(data));
  });
});
```

使用 HtmlEncoder 类时，用户可以通过<h1>元素输入 http://localhost:24897/echoenc?x=<h1>this gets converted</h1>。因此，在浏览器中显示<h1>this gets converted</ h1>。<字符编码为<，因此显示为文本。完整的编码字符串为:

```
&lt;h1&gt;this gets converted&lt;/h1&gt;
```

类似地，脚本元素会被转换，不会以脚本的形式在浏览器中运行。

**注意:**

可以使用 HtmlEncoder 类来允许检查特定的输入——例如，可能允许用户添加<b>元素。可以使用 HtmlEncoder.Create 方法通过已接受的输入创建编码器。现在的首选方法是允许用户使用 Markdown 并将 Markdown 转换为 HTML，进行一些格式化。关于 Markdown 的博客文章可以在 https://csharp.christiannagel.com/2016/07/03/markdown/上阅读。

到目前为止，示例代码已经使用了 ASP.NET Core 功能。直接从 ASP.NET Core MVC 控制器或视图内部返回一个字符串时，编码是默认进行的。需要进行额外的投资，以避免对这里的结果进行编码。

通过 ASP.NET Core 控制器只返回一个字符串，会得到一个编码的字符串(代码文件 ASPNETCoreMVCSecurity/Controllers/HomeController.cs):

```
public string Echo(string x) => x;
```

要发送未编码的字符串，可以使用 Controller 基类的 Content 方法，并指定内容返回为 text/html:

```
public IActionResult EchoUnencoded(string x) => Content(x, "text/html");
```

下面在视图中进一步使用 Razor 代码。这里，EchoWithView 方法使用 ViewBag.SampleData 把用户的输入

数据传递给视图 (代码文件 ASPNETCoreMVCSecurity/Controllers/HomeControll.cs)：

```
public IActionResult EchoWithView(string x)
{
  ViewBag.SampleData = x;
  return View();
}
```

在视图中，编码是默认进行的。使用 Razor 表达式@data 传递输入数据。data 是一个本地变量，其中，指定了传递的 ViewBag 信息。为了不使用编码，使用 Html 辅助类的 Raw 方法(代码文件 ASPNETCoreMVCSecurity/Views/Home/EchoWithView.cshtml)：

```
@{
  string data = ViewBag.SampleData;
}
<div>
  this is encoded
</div>
<div>@data</div>

<br />
<div>
  This is not encoded
</div>
<div>
  @Html.Raw(@data)
</div>
```

**注意：**

当显式向客户端发送未编码数据时，需要确保输入是可信的——例如，HTML 从 Markdown 转换而来，而不是直接返回用户输入。

**注意：**

在使用 URL 字符串的用户输入时，可以使用 UrlEncoder 类，这类似于使用用户输入作为 HTML 内容时使用 HtmlEncoder 类的方式。

## 24.6.2 SQL 注入

Web 应用程序的另一个常见问题是 SQL 注入。与 HTML 编码一样，使用内置的功能很容易避免这个问题。

下面的代码片段创建了一个 SQL 字符串，该字符串直接在 SqlSample 控制器方法中分配输入参数。有了它，用户就可以输入;SELECT * FROM Table Users，所有这些信息都显示给用户：

```
public IActionResult SqlSample(string id)
{
  string connectionString = GetConnectionString();
  var sqlConnection = new SqlConnection(connectionString);
  SqlCommand command = sqlConnection.CreateCommand();
  // don't do this - string concatenation for SQL commands!
  command.CommandText = "SELECT * FROM Customers WHERE City = " + id;
  sqlConnection.Open();
  using (SqlDataReader reader =
    command.ExecuteReader(System.Data.CommandBehavior.CloseConnection))
  {
    var sb = new StringBuilder();
    while (reader.Read())
    {
      for (int i = 0; i < reader.FieldCount; i++)
      {
        sb.Append(reader[i]);
      }
      sb.AppendLine();
    }
    ViewBag.Data = sb.ToString();
  }
  return View(); }
```

千万不要对 SQL 语句使用字符串连接。相反，使用参数或隐式地使用 Entity Framework Core 参数可以轻松

地避免这个问题。

> **注意：**
> 使用 SQL 命令参见第 25 章。Entity Framework Core 参见第 26 章。

### 24.6.3 跨站点请求伪造

跨站点请求伪造(Cross-Site Request Forgery，CSRF)是一种攻击，恶意网站试图在用户不知情的情况下重播用户并输入数据。

在下一个示例中，用户在表格中输入图书信息。Book 是一个包含 Title 和 Publisher 属性的简单模型类。在 HomeController 中，EditBook 方法返回一个视图(代码文件 ASPNETCoreMVCSecurity/Controllers/HomeController.cs)：

```
public IActionResult EditBook() => View();
```

视图定义了简单的输入数据，用户可以在其中输入 Title 和 Publisher 信息，并通过 HTTP POST 请求将这些信息传递给服务器(代码文件 ASPNETCoreMVCSecurity/Views/EditBook.cshtml)：

```
@{
    ViewData["Title"] = "EditBook";
}
<h2>Edit Book</h2>

<form asp-controller="Home" asp-action="EditBook" method="post">
  <label for="title">Title:</label>
  <input type="text" id="title" name="title" />
  <br />
  <label for="publisher">Publisher:</label>
  <input type="text" id="publisher" name="publisher" />
  <br />
  <input type="submit" value="Submit" />
</form>
```

使用 HTTP POST 请求，将调用下面的 EditBook 方法，来显示带有输入用户数据的视图(代码文件 ASPNETCoreMVCSecurity/Controllers/HomeController.cs)：

```
[HttpPost]
public IActionResult EditBook(Book book) => View("EditBookResult", book);
```

打开 URL http://localhost:24897/Home/EditBook，运行应用程序时，可以输入图书信息，单击 submit 按钮，从控制器中接收信息，图书信息显示在视图结果中。

与此同时，恶意网站只需要使用相同的链接，以自己的形式发布数据。检查以下代码片段，其中的表单元素引用了与以前相同的 URL。此表单托管于另一个网站 http//localhost:9817/dothis.html。这只是一个不同的端口，但也可以是不同的域名。用户不需要在表单中输入任何内容(输入元素是隐藏的，因此不会显示给用户)。用户只需要单击 submit 按钮，不需要知道幕后发生了什么不同的事情(代码文件 HackingSite/wwwroot/dothis.html)：

```
<h1>Click this for a win!</h1>

<!-- form has a redirect to the website being hacked -->
<form action="http://localhost:24897/Home/EditBook" method="post">
  <input type="hidden" value="bad book title" name="title" />
  <input type="hidden" value="bad publisher" name="publisher" />
  <input type="submit" value="Click Now!" />
</form>
```

单击这个链接时，恶意数据会以用户的名义传送到网站。如果用户通过 Book 网站进行了身份验证，并且没有注销，数据就以用户的名义提交，并且很可能在不同的交付地址下了一些订单。

为了避免这种行为，ASP.NET Core 提供了反伪造令牌。这样的令牌需要从用户应该使用的表单中创建，以输入有效数据，并在接收数据时进行验证。

编辑图书表单现在通过 HTML 辅助方法 AntiForgeryToken 改为包含该令牌(代码文件 ASPNETCoreMVCSecurity/Views/EditBook.cshtml)：

```
<form asp-controller="Home" asp-action="EditBook" method="post">
  @Html.AntiForgeryToken()
```

```
    <label for="title">Title:</label>
    <input type="text" id="title" name="title" />
    <br />
    <label for="publisher">Publisher:</label>
    <input type="text" id="publisher" name="publisher" />
    <br />
    <input type="submit" value="Submit" />
  </form>
```

运行应用程序时，可以看到一个隐藏表单字段，其中包含自动生成的令牌。当检索数据时，使用 ValidateAntiForgeryToken 属性对标记进行验证(代码文件 ASPNETCoreMVCSecurity/Controllers/HomeController.cs)：

```
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult EditBook(Book book) => View("EditBookResult", book);
```

现在运行恶意网站时，将返回响应，而不接受无效数据。

## 24.7　小结

本章讨论了与.NET 应用程序相关的几个安全性方面。用户用标识和主体表示，这些类实现了 IIdentity 和 IPrincipal 接口。还介绍了如何访问标识中的声称。

本章介绍了加密方法，说明了数据的签名和加密，以安全的方式交换密钥。.NET 提供了对称加密算法和非对称加密算法，以及散列和签名。

使用访问控制列表还可以读取和修改对操作系统资源(如文件)的访问权限。ACL 的编程方式与安全管道、注册表键、Active Directory 项以及许多其他操作系统资源的编程方式相同。

在许多情况下，可以在较高的抽象级别上处理安全性。例如，使用 HTTPS 访问 Web 服务器，在后台交换加密密钥。File 类提供了 Encrypt 方法(使用 NTFS 文件系统)，轻松地加密文件。知道这个功能如何发挥作用也很重要。

对于 Web 应用程序，讨论了因为信任用户输入而导致各种攻击所带来的常见问题，包括跨站点的请求伪造，也探讨了如何使用编码来避免各种问题，如何使用反伪造请求令牌来避免 XSRF。

接下来的两章使用数据库中的数据，从 ADO.NET 开始讨论。

第 25 章

# ADO.NET 和事务

**本章要点**

- 连接数据库
- 执行命令
- 调用存储过程
- ADO.NET 对象模型
- 用 ADO.NET 完成事务
- 用 System.Transactions 管理事务

**本章源代码下载地址(wrox.com)：**

打开 www.wrox.com 的 Download Code 选项卡可下载本章源代码。源代码也可以在 ADONET 目录的 https://github.com/ProfessionalCSharp/ProfessionalCSharp7 中找到。本章代码分为以下几个主要的示例文件：

- ConnectionSamples
- CommandSamples
- AsyncSamples
- TransactionSamples
- SystemTransactionSamples

## 25.1 ADO.NET 概述

本章讨论如何使用 ADO.NET 访问 C#程序中的关系数据库，例如 SQL Server，主要介绍如何连接数据库，以及断开与数据库的连接。如何使用查询，如何添加和更新记录。学习各种命令对象选项，了解如何为 SQL Server 提供程序类提供的每个选项使用命令；如何通过命令对象调用存储过程，以及如何使用事务。

ADO.NET 之前使用 OLEDB 和 ODBC 附带了不同的数据库提供程序，一个提供程序用于 SQL Server；另一个提供程序用于 Oracle。OLEDB 技术不再获得支持，所以这个提供程序不应该用于新的应用程序。对于访问 Oracle 数据库，微软的提供程序也不再使用，因为来自 Oracle(http://www.oracle.com/technetwork/topics/dotnet/)的提供程序能更好地满足需求。对其他数据源(也用于 Oracle)，有许多可用的第三方提供程序。使用 ODBC 提供程序之前，应该给所访问的数据源使用专用的提供程序。本章中的代码示例基于 SQL Server，但也可以把它

改为使用不同的连接和命令对象，如访问 Oracle 数据库时，使用 OracleConnection 和 OracleCommand，而不是使用 SqlConnection 和 SqlCommand。

> **注意：**
> 本章不介绍把表放在内存中的 DataSet，尽管.NET Core 2.0 支持它们。DataSet 允许从数据库中检索记录，并把内容存储在内存的数据表关系中。DataSet 在.NET 的早期版本中用得很多。现在可以使用 Entity Framework Core（EF Core），参见第 26 章。这个新技术允许使用对象关系，而不是使用基于表的关系。

## 25.1.1　示例数据库

本章的示例使用 Books 数据库。这个数据库的备份文件位于 CreateDatabase 目录下的源代码中。使用该备份文件，可以通过 SQL Server Management Studio 恢复数据库备份，如图 25-1 所示。另外，也可以使用 SQL 脚本 CreateBooks.sql 创建数据库。如果系统上没有 SQL Server Management Studio，则可以从 https://docs.microsoft.com/sql/ssms/download-sql-server-management-studio-ssms 上下载一个免费的版本。

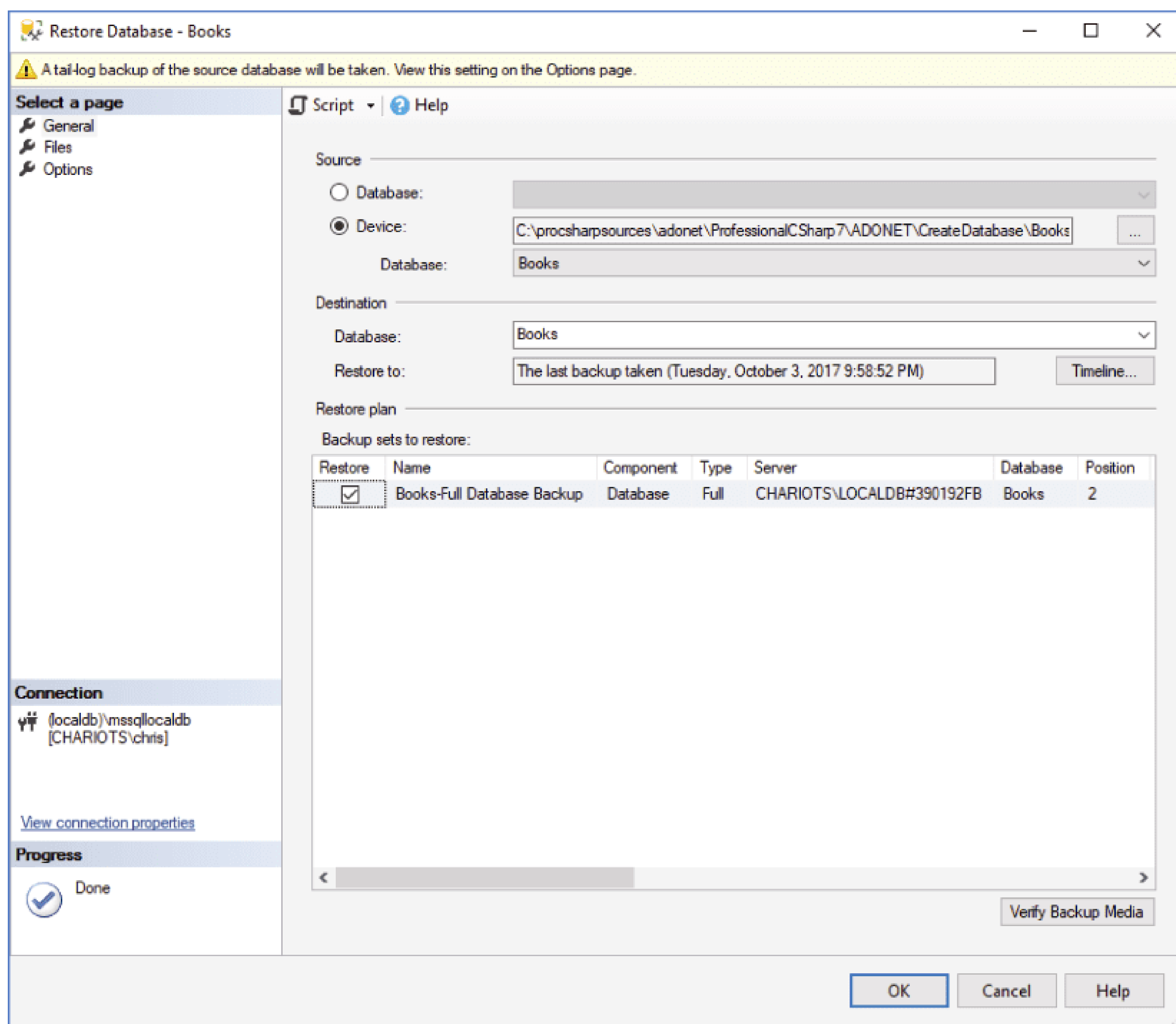



图 25-1

本章使用的 SQL Server 是 SQL Server LocalDb。这个数据库服务器安装为 Visual Studio 的一部分。也可以使用任何其他 SQL Server 版本，只需要改变相应的连接字符串。

## 25.1.2　NuGet 包和名称空间

ADO.NET 示例的示例代码利用以下依赖项和名称空间：

**依赖项**

Microsoft.Extensions.Configuration

Microsoft.Extensions.Configuration.Json

System.Data.SqlClient

**名称空间**

Microsoft.Extensions.Configuration

System

System.Collections

System.Data

System.Data.SqlClient

System.IO

System.Threading.Tasks

## 25.2 使用数据库连接

为了访问数据库，需要提供某种连接参数，如运行数据库的计算机和登录证书。使用 SqlConnection 类连接 SQL Server。

下面的代码段说明了如何创建、打开和关闭 Books 数据库的连接(代码文件 ConnectionSamples/Program.cs)。

```
public static void OpenConnection()
{
  string connectionString = @"server=(localdb)\MSSQLLocalDB;" +
    "integrated security=SSPI;database=Books";

  var connection = new SqlConnection(connectionString);
  connection.Open();
  // Do something useful
  Console.WriteLine("Connection opened");
  connection.Close();
}
```

**注意：**

SqlConnection 类实现了 IDisposable 接口，其中包含 Dispose 方法和 Close 方法。这两个方法的功能相同，都是释放连接。这样，就可以使用 using 语句来关闭连接。

在该示例的连接字符串中，使用的参数如下所示。连接字符串中的参数用分号分隔开。

- server=(localdb)\ MSSQLLocalDB——表示要连接到的数据库服务器。SQL Server 允许在同一台计算机上运行多个不同的数据库服务器实例，这里连接到 localdb 服务器和安装 SQL Server 时创建的 SQL Server 实例 MSSQLLocalDB。如果使用的是本地安装的 SQL Server，就把这一部分改为 server=(local)。如果不使用关键字 server，还可以使用 Data Source。要连接到 SQL Azure，可以设置 Data Source=servername.database.windows.net。
- database=Books——这描述了要连接到的数据库实例。每个 SQL Server 进程都可以提供几个数据库实例。如果不使用关键字 database，可以使用 Initial Catalog。
- integrated security=SSPI——这个参数使用 Windows Authentication 连接到数据库，如果使用 SQL Azure，就需要设置 User Id 和 Password 或者使用 Azure Active Directory。

**注意：**

在 http://www.connectionstrings.com 上可以找到许多不同数据库的连接字符串信息。

这个 ConnectionSamples 示例使用定义好的连接字符串打开数据库连接，再关闭该连接。一旦打开连接后，就可以对数据源执行命令，完成后，就可以关闭连接。

### 25.2.1　管理连接字符串

不在 C#代码中硬编码连接字符串，而是最好从配置文件中读取它。在.NET Core 中，配置文件可以是 JSON 或 XML 格式，或从环境变量中读取。在下面的示例中，连接字符串从一个 JSON 配置文件中读取(代码文件 ConnectionSamples/config.json)：

```
{
  "Data": {
    "DefaultConnection": {
    "ConnectionString":
      "Server=(localdb)\\MSSQLLocalDB;Database=Books;
        Trusted_Connection=True;"
    }
  }
}
```

使用 NuGet 包 Microsoft.Extensions.Configuration 定义的 Configuration API 可以读取 JSON 文件。为了使用 JSON 配置文件，还要添加 NuGet 包 Microsoft.Extensions.Configuration.Json。为了读取配置文件，创建 ConfigurationBuilder。AddJsonFile 扩展方法添加 JSON 文件 config.Json，从这个文件中读取配置信息——假定它与程序在相同的路径中。要配置另一条路径，可以调用 SetBasePath 方法。调用 ConfigurationBuilder 的 Build 方法，从所有添加的配置文件中构建配置，返回一个实现了 IConfiguration 接口的对象。这样，就可以检索配置值，如 Data:DefaultConnection:ConnectionString 的配置值(代码文件 ConnectionSamples/Program.cs)：

```
public static void ConnectionUsingConfig()
{
  var configurationBuilder = new ConfigurationBuilder()
    .SetBasePath(Directory.GetCurrentDirectory())
    .AddJsonFile("config.json");
  IConfiguration config = configurationBuilder.Build();
  string connectionString = config["Data:DefaultConnection:ConnectionString"];
  Console.WriteLine(connectionString);
}
```

### 25.2.2　连接池

几年前实现两层应用程序时，是在应用程序启动时打开连接，只有在关闭应用程序时才关闭连接。现在就不用这么做。使用这个程序架构的原因是，需要一定的时间来打开连接。现在，关闭连接不会关闭与服务器的连接。相反，连接会添加到连接池中。再次打开连接时，它可以从池中提取，因此打开连接会非常快速，只有第一次打开连接需要一定的时间。

连接池可以用几个选项在连接字符串中配置。选项 Pooling 设置为 false，会禁用连接池；它默认为启用：Pooling = true。Min Pool Size 和 Max Pool Size 允许配置池中的连接数。默认情况下，Min Pool Size 的值为 0，Max Pool Size 的值为 100。Connection Lifetime 定义了连接在释放前在池中保持不活跃状态的时间。

### 25.2.3　连接信息

在创建连接之后，可以注册事件处理程序，来获得一些连接信息。SqlConnection 类定义了 InfoMessage 和 StateChange 事件。每次从 SQL Server 返回一个信息或警告消息时，就触发 InfoMessage 事件。连接的状态变化时，就触发 StateChange 事件，例如打开或关闭连接(代码文件 ConnectionSamples/Program.cs)：

```
public static void ConnectionInformation()
{
  using (var connection = new SqlConnection(GetConnectionString()))
  {
    connection.InfoMessage += (sender, e) =>
    {
      Console.WriteLine($"warning or info {e.Message}");
    };

    connection.StateChange += (sender, e) =>
    {
      Console.WriteLine($"current state: {e.CurrentState}, before: " +
```

```
        $"{e.OriginalState}");
    };
    try
    {
      connection.StatisticsEnabled = true;
      connection.FireInfoMessageEventOnUserErrors = true;
      connection.Open();

      Console.WriteLine("connection opened");
      //... Read some records
    }
    catch (Exception ex)
    {
      Console.WriteLine(ex.Message);
    }
  }
}
```

运行应用程序时，会触发 StateChange 事件，看到 Open 和 Closed 状态：

```
current state: Open, before: Closed
connection opened
... output reading records
current state: Closed, before: Open
```

如果出现了异常，默认不触发 InfoMessage 事件。设置 FireInfoMessageEventOnUserErrors 属性，可以改变这个行为。这样在出错时，例如使用了 Titl 而不是 Title，就可以从应用程序中看到这个信息：

```
current state: Open, before: Closed
connection opened
warning or info: Invalid column name 'Titl'.
... output reading records
current state: Closed, before: Open
```

SqlConnection 类还提供了统计信息。只需要设置 StatisticsEnabled 属性，就可以在 RetrieveStatistics 方法中检索统计信息。这个方法通过实现了 IDictionary 接口的对象返回统计信息：

```
IDictionary statistics = connection.RetrieveStatistics();
ShowStatistics(statistics);
connection.ResetStatistics();
```

ShowStatistics 方法迭代接收了 IDictionary 接口的所有键，并显示所有值：

```
private static void ShowStatistics(IDictionary statistics)
{
  Console.WriteLine("Statistics");
  foreach (var key in statistics.Keys)
  {
    Console.WriteLine($"{key}, value: {statistics[key]}");
  }
  Console.WriteLine();
}
```

从 Books 表中检索所有记录，就会显示连接中的这些统计信息：

```
BuffersReceived, value: 1
BuffersSent, value: 1
BytesReceived, value: 142
BytesSent, value: 124
CursorOpens, value: 0
IduCount, value: 0
IduRows, value: 0
PreparedExecs, value: 0
Prepares, value: 0
SelectCount, value: 0
SelectRows, value: 0
ServerRoundtrips, value: 1
SumResultSets, value: 0
Transactions, value: 0
UnpreparedExecs, value: 1
ConnectionTime, value: 140
ExecutionTime, value: 456
NetworkServerTime, value: 8
```

# 25.3　命令

25.2 节“使用数据库连接”简要介绍了针对数据库执行的命令。简言之，命令就是一个要在数据库上执行的包含 SQL 语句的文本字符串。命令也可以是一个存储过程，如本节后面所述。

把 SQL 子句作为一个参数传递给 Command 类的构造函数，就可以构造一条命令，如下例所示(代码文件 CommandSamples/Program.cs )：

```
public static void CreateCommand()
{
  using (var connection = new SqlConnection(GetConnectionString()))
  {
    string sql = "SELECT [Title], [Publisher], [ReleaseDate] " +
      "FROM [ProCSharp].[Books]";
    var command = new SqlCommand(sql, connection);
    connection.Open();
    //...
  }
}
```

通过调用 SqlConnection 的 CreateCommand 方法，把 SQL 语句赋予 CommandText 属性，也可以创建命令：

```
SqlCommand command = connection.CreateCommand();
command.CommandText = sql;
```

命令通常需要参数。例如，下面的 SQL 语句需要一个 Title 参数。不要试图使用字符串连接来建立参数。相反，总是应使用 ADO.NET 的参数特性：

```
string sql = "SELECT [Title], [Publisher], [ReleaseDate] " +
"FROM [ProCSharp].[Books] WHERE lower([Title]) LIKE @Title";
var command = new SqlCommand(sql, connection);
```

将参数添加到 SqlCommand 对象中时，有一个简单的方式可以使用 Parameters 属性返回 SqlParameterCollection 和 AddWithValue 方法：

```
command.Parameters.AddWithValue("Title", "Professional C#%");
```

有一个更有效的方式，但需要更多的编程工作：通过传递名称和 SQL 数据类型，使用 Add 方法的重载版本：

```
command.Parameters.Add("TitleStart", SqlDbType.NVarChar, 50);
command.Parameters["Title"].Value = "Professional C#%";
```

也可以创建一个 SqlParameter 对象，并添加到 SqlParameterCollection 中。

**注意：**

不要试图给 SQL 参数使用字符串连接。它经常被用于 SQL 注入攻击。使用 SqlParameter 对象会抑制这种攻击。

定义好命令后，就需要执行它。执行语句有许多方式，这取决于要从命令中返回什么数据。SqlCommand 类提供了下述可执行的命令：

- ExecuteNonQuery()——执行命令，但不返回任何结果。
- ExecuteReader()——执行命令，返回一个类型化的 IDataReader。
- ExecuteScalar()——执行命令，返回结果集中第一行第一列的值。

## 25.3.1　ExecuteNonQuery()方法

这个方法一般用于 UPDATE、INSERT 或 DELETE 语句，其中唯一的返回值是受影响的记录个数。但如果调用带输出参数的存储过程，该方法就有返回值。示例代码在 ProCSharp.Books 表中创建了一个新的记录。图 25-2 显示 Visual Studio 2017 中这个表的设计视图。

这个表把 Id 作为主键，Id 是一个标识列，因此创建记录时不需要提供它。Title 和 Isbn 列都不允许空值(见

图 25-2)，但其他列允许。在示例代码中，所有列都填充了值。ExecuteNonQuery 方法定义了 SQL INSERT 语句，添加了参数值，并调用 SqlCommand 类的 ExecuteNonQuery 方法(代码文件 CommandSamples/Program.cs)：

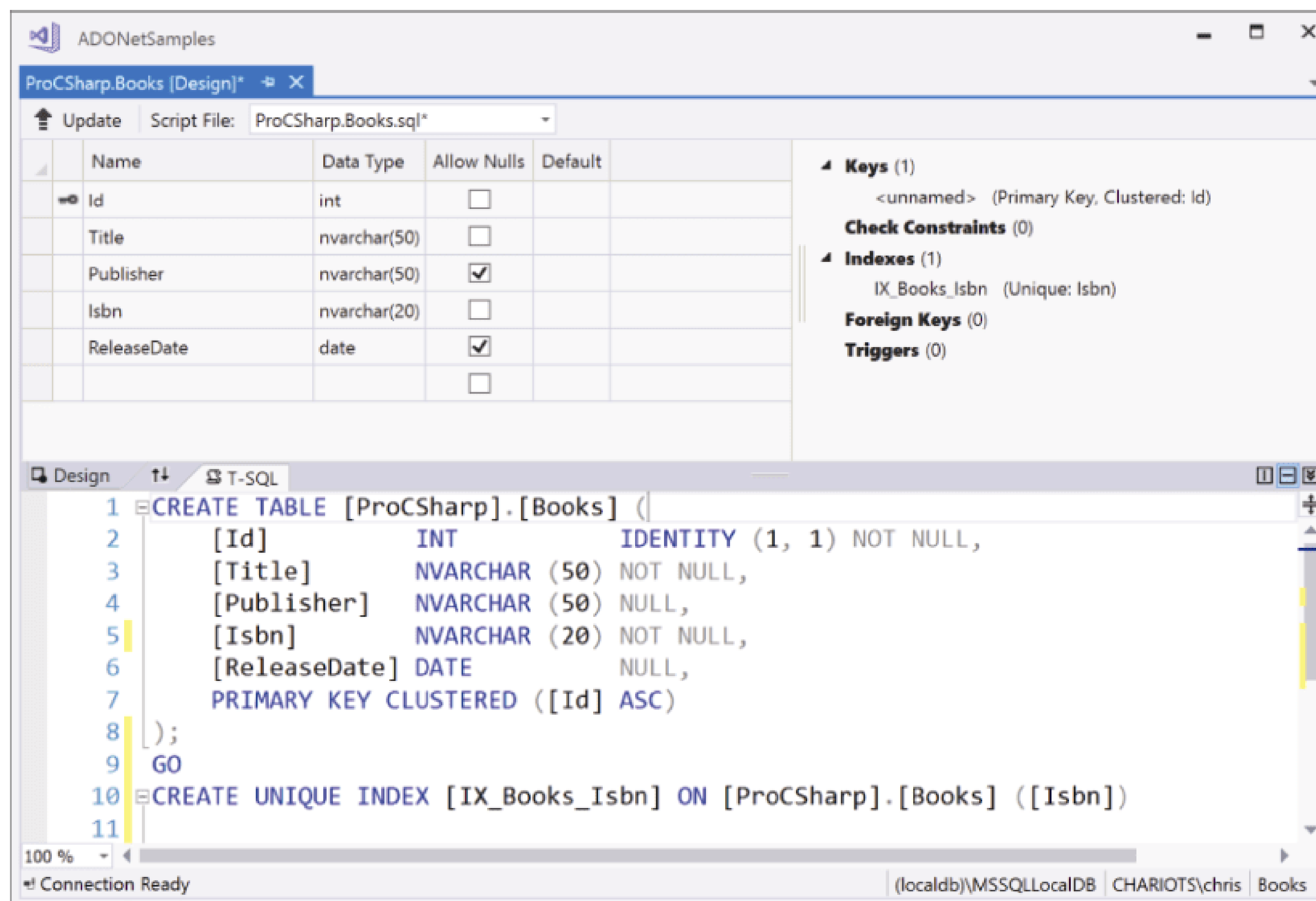



图 25-2

```
public static void ExecuteNonQuery
{
  try
  {
    using (var connection = new SqlConnection(GetConnectionString()))
    {
      string sql = "INSERT INTO [ProcSharp].[Books] " +
        "([Title], [Publisher], [Isbn], [ReleaseDate]) " +
        "VALUES (@Title, @Publisher, @Isbn, @ReleaseDate)";
      var command = new SqlCommand(sql, connection);
      command.Parameters.AddWithValue("Title",
        "Professional C# 7 and .NET Core 2.0");
      command.Parameters.AddWithValue("Publisher", "Wrox Press");
      command.Parameters.AddWithValue("Isbn", "978-1119449270");
      command.Parameters.AddWithValue("ReleaseDate", new DateTime(2018, 4, 2));

      connection.Open();
      int records = command.ExecuteNonQuery();
      Console.WriteLine($"{records} record(s) inserted");
    }
  }
  catch (SqlException ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

ExecuteNonQuery()方法返回命令所影响的行数，它是一个整数。第一次运行这个方法时，插入了一个记录。第二次运行相同的方法时，会得到一个异常，因为唯一索引有冲突。Isbn 定义了唯一的索引，只允许使用一次。第二次运行该方法时，需要先删除前面创建的记录。

## 25.3.2 ExecuteScalar()方法

在许多情况下，需要从 SQL 语句返回一个结果，如给定表中的记录个数，或者服务器上的当前日期/时间。ExecuteScalar()方法就可以用于这些场合：

```
public static void ExecuteScalar()
```

```
{
  using (var connection = new SqlConnection(GetConnectionString()))
  {
    string sql = "SELECT COUNT(*) FROM [ProCSharp].[Books]";
    SqlCommand command = connection.CreateCommand();
    command.CommandText = sql;
    connection.Open();
    object count = command.ExecuteScalar();
    Console.WriteLine($"counted {count} book records");
  }
}
```

该方法返回一个对象，根据需要，可以把该对象强制转换为合适的类型。如果所调用的 SQL 只返回一列，则最好使用 ExecuteScalar()方法来检索这一列。这也适合于只返回一个值的存储过程。

### 25.3.3　ExecuteReader()方法

ExecuteReader()方法执行命令，并返回一个 DataReader 对象，返回的对象可以用于遍历返回的记录。ExecuteReader 示例使用一个 SQL 查询，它在 ExecuteReader 方法中通过本地函数 GetBookQuery 定义 (代码文件 CommandSamples/Program.cs)：

```
public static void ExecuteReader(string titleParameter)
{
  string GetBookQuery() =>
    "SELECT [Id], [Title], [Publisher], [ReleaseDate] "+
    "FROM [ProCSharp].[Books] WHERE lower([Title]) LIKE @Title " +
    "ORDER BY [ReleaseDate] DESC";
  //...
}
```

**注意：**
本地函数详见第 13 章。

当调用 SqlCommand 对象的 ExecuteReader 方法时，返回 SqlDataReader。注意，SqlDataReader 使用后需要销毁。还要注意，这次 SqlConnection 对象没有在方法的最后明确地销毁。给 ExecuteReader 方法传递参数 CommandBehavior.CloseConnection，会在关闭读取器时，自动关闭连接。如果没有提供这个设置，就仍然需要关闭连接。

从数据读取器中读取记录时，Read 方法在 while 循环中调用。Read 方法的第一个调用将光标移动到返回的第一条记录上。再次调用 Read 时，光标定位到下一个记录(只要还有记录)。如果下一步位置上没有记录了，Read 方法就返回 false。访问列的值时，调用不同的 GetXXX 方法，如 GetInt32、GetString 和 GetDateTime。这些方法是强类型化的，因为它们返回所需的特定类型，如 int、string 和 DateTime。传递给这些方法的索引对应于用 SQL SELECT 语句检索的列，因此即使数据库结构有变化，该索引也保持不变。在强类型化的 GetXXX 方法中，需要注意从数据库返回的 null 值；此时，GetXXX 方法会抛出一个异常。对于检索的数据，只有 ReleaseDate 可以为空。在这种情况下为了避免异常，使用 C#条件语句?:和 SqlDataReader.IsDbNull 方法，检查值是否是 null。如果是，就把 null 分配给可空的 DateTime。只有值不是 null，才使用 GetDateTime 方法访问 DateTime (代码文件 CommandSamples/Program.cs)：

```
public static void ExecuteReader(string title)
{
  //...

  var connection = new SqlConnection(GetConnectionString());
  var command = new SqlCommand(GetBookQuery(), connection);
  var parameter = new SqlParameter("Title", SqlDbType.NVarChar, 50)
  {
    Value = $"{title}%"
  };
  command.Parameters.Add(parameter);

  connection.Open();
  using (SqlDataReader reader =
    command.ExecuteReader(CommandBehavior.CloseConnection))
  {
```

```
      while (reader.Read())
      {
        int id = reader.GetInt32(0);
        string bookTitle = reader.GetString(1);
        string publisher = reader.GetString(2);
        DateTime? releaseDate =
          reader.IsDbNull(3) ? (DateTime?)null : reader.GetDateTime(3);
        DateTime from = reader.GetDateTime(4);
        Console.WriteLine($"{id,5}. {bookTitle,-40} {publisher,-15} " +
          $"{releaseDate:d}");
      }
    }
  }
```

运行应用程序，把标题 Professional C#传递给 ExecuteReader 方法，输出如下：

```
1015. Professional C# 7 and .NET Core 2.0      Wrox Press     4/2/2018
   1. Professional C# 6 and .NET Core 1.0      Wrox Press     4/11/2016
   2. Professional C# 5.0 and .NET 4.5.1       Wrox Press     2/9/2014
  11. Professional C# 2012 and .NET 4.5        Wrox Press     10/18/2012
   9. Professional C# 4 and .NET 4             Wrox Press     3/8/2010
   8. Professional C# 2008                     Wrox Press     5/24/2008
  18. Professional C# 2005 with .NET 3.0       Wrox Press     6/12/2007
  12. Professional C# 2005                     Wrox Press     11/7/2005
   6. Professional C# 3rd Edition              Wrox Press     6/2/2004
  10. Professional C# 2nd Edition              Wrox Press     3/28/2002
 010. Professional C# Web Services             Wrox Press     12/1/2001
1012. Professional C# (Beta 2 Edition)         Wrox Press     6/1/2001
```

对于 SqlDataReader，不是使用类型化的 GetXXX 方法，而可以使用无类型的索引器返回一个对象。为此，需要转换为相应的类型：

```
int id = (int)reader[0];
string bookTitle = (string)reader[1];
string publisher = (string)reader[2];
DateTime? releaseDate = (DateTime?)reader[3];
```

SqlDataReader 的索引器还允许使用 string 而不是 int 传递列名。在这些不同的选项中，这是最慢的方法，但它可以满足需求。与发出服务调用所需的时间相比，访问索引器所需的额外时间可以忽略不计：

```
int id = (int)reader["Id"];
string bookTitle = (string)reader["Title"];
string publisher = (string)reader["Publisher"];
DateTime? releaseDate = (DateTime?)reader["ReleaseDate"];
```

### 25.3.4 调用存储过程

用命令对象调用存储过程，就是定义存储过程的名称，给过程的每个参数添加参数定义，然后用上一节中给出的其中一种方法执行命令。

下面的示例调用存储过程 GetBooksByPublisher，得到一家出版社的所有图书。这个存储过程接收一个参数，使用递归查询返回所请求的所有图书的记录：

```
CREATE PROCEDURE [ProCSharp].[GetBooksByPublisher]
  @publisher nvarchar(50)
AS
  SELECT [Id], [Title], [Publisher], [ReleaseDate] FROM [ProCSharp].[Books]
  WHERE [Publisher] = @publisher ORDER BY [ReleaseDate]
```

为了调用存储过程，SqlCommand 对象的 CommandText 设置为存储过程的名称，CommandType 设置为 CommandType.StoredProcedure。除此之外，该命令的调用类似于以前的方式。参数使用 SqlCommand 对象的 CreateParameter 方法创建，也可以使用其他方法创建之前使用的参数。对于参数，填充 SqlDbType、ParameterName 和 Value 属性。因为存储过程返回记录，所以它通过调用方法 ExecuteReader 来调用(代码文件 CommandSamples/Program.cs)：

```
private static void StoredProcedure(string publisher)
{
  using (var connection = new SqlConnection(GetConnectionString()))
  {
    SqlCommand command = connection.CreateCommand();
    command.CommandText = "[ProCSharp].[GetBooksByPublisher]";
    command.CommandType = CommandType.StoredProcedure;
```

```
        SqlParameter p1 = command.CreateParameter();
        p1.SqlDbType = SqlDbType.NvarChar;
        p1.ParameterName = "@Publisher";
        p1.Value = publisher;
        command.Parameters.Add(p1);
        connection.Open();
        using (SqlDataReader reader = command.ExecuteReader())
        {
          while (reader.Read())
          {
            int recursionLevel = (int)reader["Id"];
            string title = (string)reader["Title"];
            string pub = (string)reader["Publisher"];
            DateTime releaseDate = (DateTime)reader["ReleaseDate"];
            Console.WriteLine($"{title} - {pub}; {releaseDate:d}");
          }
        }
      }
    }
```

运行应用程序，传递出版社 Wrox Press，得到如下所示的结果：

```
Professional C# (Beta 2 Edition) - Wrox Press; 6/1/2001
Beginning C# - Wrox Press; 9/15/2001
Professional C# Web Services - Wrox Press; 12/1/2001
Professional C# 2nd Edition - Wrox Press; 3/28/2002
Beginning Visual C# - Wrox Press; 8/20/2002
Professional .NET Network Programming - Wrox Press; 10/1/2002
ASP to ASP.NET Migration Handbook - Wrox Press; 2/1/2003
Professional C# 3rd Edition - Wrox Press; 6/2/2004
Professional C# 2005 - Wrox Press; 11/7/2005
Beginning Visual C# 2005 - Wrox Press; 11/7/2005
Professional C# 2005 with .NET 3.0 - Wrox Press; 6/12/2007
Beginning Visual C# 2008 - Wrox Press; 5/5/2008
Professional C# 2008 - Wrox Press; 5/24/2008
Professional C# 4 and .NET 4 - Wrox Press; 3/8/2010
Beginning Visual C# 2010 - Wrox Press; 4/5/2010
Real World .NET, C#, and Silverlight - Wrox Press; 11/22/2011
Professional C# 2012 and .NET 4.5 - Wrox Press; 10/18/2012
Beginning Visual C# 2012 Programming - Wrox Press; 12/4/2012
Professional C# 5.0 and .NET 4.5.1 - Wrox Press; 2/9/2014
Professional C# 6 and .NET Core 1.0 - Wrox Press; 4/11/2016
Professional C# 7 and .NET Core 2.0 - Wrox Press; 4/2/2018
```

根据存储过程返回的内容，需要用 ExecuteReader、ExecuteScalar 或 ExecuteNonQuery 调用存储过程。

对于包含 Output 参数的存储过程，需要指定 SqlParameter 的 Direction 属性。默认情况下，Direction 是 ParameterDirection. Input：

```
var pOut = new SqlParameter();
pOut.Direction = ParameterDirection.Output;
```

## 25.4　异步数据访问

访问数据库可能要花一些时间。这里不应该阻塞用户界面。ADO.NET 类通过异步方法和同步方法提供了基于任务的异步编程。下面的代码片段类似于上一个使用 SqlDataReader 的代码，但它使用了异步的方法调用。连接用 SqlConnection. OpenAsync 打开，读取器从 SqlCommand.ExecuteReaderAsync 方法中返回，记录使用 SqlDataReader.ReadAsync 检索。在所有这些方法中，调用线程没有阻塞，但是可以在得到结果前，执行其他操作(代码文件 AsyncSamples/Program.cs)：

```
public static async Task Main()
{
  await ReadAsync("Wrox Press");
}

public static async Task ReadAsync(int productId)
{
  var connection = new SqlConnection(GetConnectionString());

  string sql = "SELECT [Title], [Publisher], [ReleaseDate] " +
    "FROM [ProCSharp].[Books] WHERE lower([Title]) " +
    "LIKE @Title ORDER BY [ReleaseDate]";
```

```
    var command = new SqlCommand(sql, connection);
    var titleParameter = new SqlParameter("Title", SqlDbType.NVarChar, 50);
    titleParameter.Value = title;
    command.Parameters.Add(titleParameter);

    await connection.OpenAsync();

    using (SqlDataReader reader =
      await command.ExecuteReaderAsync(CommandBehavior.CloseConnection))
    {
      while (await reader.ReadAsync())
      {
        int id = reader.GetInt32(0);
        string bookTitle = reader.GetString(1);
        string publisher = reader[2].ToString();
        DateTime? releaseDate =
          reader.IsDBNull(3) ? (DateTime?)null : reader.GetDateTime(3);
        Console.WriteLine($"{id,5}. {bookTitle,-40} {publisher,-15} " +
          $"{releaseDate:d}");
      }
    }
  }
```

**注意：**
异步 Main()方法至少需要 C# 7.1。

使用异步方法调用，不仅有利于 Windows 应用程序，也有利于在服务器端同时进行多个调用。ADO.NET API 的异步方法有重载版本来支持 CancellationToken，使长时间运行的方法早些停止。

**注意：**
异步方法调用和 CancellationToken 详见第 15 章。

# 25.5 事务

默认情况下，一个命令运行在一个事务中。如果需要执行多个命令，所有这些命令都执行完毕，或都没有执行，就可以显式地启动和提交事务。

事务的特征可以用术语 ACID 来定义，ACID 是 Atomicity、Consistency、Isolation 和 Durability 的首字母缩写。

- Atomicity(原子性)——表示一个工作单元。在事务中，要么整个工作单元都成功完成，要么都不完成。
- Consistency(一致性)——事务开始前的状态和事务完成后的状态必须有效。在执行事务的过程中，状态可以有临时值。
- Isolation(隔离性)——表示并发进行的事务独立于状态，而状态在事务处理过程中可能发生变化。在事务未完成时，事务 A 看不到事务 B 中的临时状态。
- Durability(持久性)——在事务完成后，它必须以可持久的方式存储起来。如果关闭电源或服务器崩溃，该状态在重新启动时必须恢复。

**注意：**
事务和有效状态很容易用婚礼来解释。新婚夫妇站在事务协调员面前，事务协调员询问一位新人：“你愿意与你身边的男人结婚吗？”如果第一位新人同意，就询问第二位新人：“你愿意与这个女人结婚吗？”如果第二位新人反对，第一位新人就接收到回滚消息。这个事务的有效状态是，要么两个人都结婚，要么两个人都不结婚。如果两个人都同意结婚，事务就会提交，这两个人就都处于已结婚的状态。如果其中一个人反对，事务就会终止，两个人都处于未结婚的状态。无效的状态是：一个人已结婚，而另一个没有结婚。事务确保结果永远不处于无效状态。

在 ADO.NET 中，通过调用 SqlConnection 的 BeginTransaction 方法就可以开始事务。事务总是与一个连接关联起来；不能在多个连接上创建事务。BeginTransaction 方法返回一个 SqlTransaction，SqlTransaction 需要使

用运行在相同事务下的命令(代码文件 TransactionSamples/Program.cs)：

```
public static void TransactionSample()
{
  using (var connection = new SqlConnection(GetConnectionString()))
  {
    await connection.OpenAsync();
    SqlTransaction tx = connection.BeginTransaction();
    //...
  }
}
```

**注意：**

为什么 OpenAsync 和 BeginTransaction 方法在 try 块之外定义？这些调用也可能失败。例如，如果 OpenAsync 方法失败，则在本地 catch 块中不会捕获异常，而是在 TransactionSample 方法的外部搜索匹配的 catch。这是单独处理这些异常的好方法。tx 变量需要在 try 块之外声明，否则不可能在 catch 中使用它。

代码示例在 ProCSharp.Books 表中创建一个记录。使用 SQL 子句 INSERT INTO 添加记录。Books 表定义了一个自动递增的标识符，它使用返回创建的标识符的第二条 SQL 语句 SELECT SCOPE_IDENTITY()返回。在实例化 SqlCommand 对象后，通过设置 Connection 属性来分配连接，设置 Transaction 属性来指定事务。在 ADO.NET 事务中，不能把事务分配给使用不同连接的命令。不过，可以用相同的连接创建与事务不相关的命令：

```
public static void TransactionSample()
{
  //...
  try
  {
    string sql = "INSERT INTO [ProcSharp].[Books] " +
      "([Title], [Publisher], [Isbn], [ReleaseDate])" +
      "VALUES (@Title, @Publisher, @Isbn, @ReleaseDate); " +
      "SELECT SCOPE_IDENTITY()";

    var command = new SqlCommand
    {
      CommandText = sql,
      Connection = connection,
      Transaction = tx
    }
    //...
}
```

在定义参数并填充值后，通过调用方法 ExecuteScalarAsync 来执行命令。这次，ExecuteScalarAsync 方法和 INSERT INTO 子句一起使用，因为完整的 SQL 语句通过返回一个结果来结束：从 SELECT SCOPE_IDENTITY()返回创建的标识符。如果在 WriteLine 方法后设置一个断点，检查数据库中的结果，在数据库中就不会看到新记录，虽然已经返回了创建的标识符。原因是事务还没有提交：

```
public static void TransactionSample()
{
  //...
  var p1 = new SqlParameter("Title", SqlDbType.NVarChar, 50);
  var p2 = new SqlParameter("Publisher", SqlDbType.NVarChar, 50);
  var p3 = new SqlParameter("Isbn", SqlDbType.NVarChar, 20);
  var p4 = new SqlParameter("ReleaseDate", SqlDbType.Date);
  command.Parameters.AddRange(new SqlParameter[] { p1, p2, p3, p4 });

  command.Parameters["Title"].Value = "Professional C# 8 and .NET Core 3.0";
  command.Parameters["Publisher"].Value = "Wrox Press";
  command.Parameters["Isbn"].Value = "42-08154711";
  command.Parameters["ReleaseDate"].Value = new DateTime(2020, 9, 2);

  object id = await command.ExecuteScalarAsync();
  Console.WriteLine($"record added with id: {id}");
  //...
}
```

现在可以在同一事务中创建另一个记录。在示例代码中，使用同样的命令，连接和事务仍然相关，只是在再次调用 ExecuteScalarAsync 前改变了值。也可以创建一个新的 SqlCommand 对象，访问同一个数据库中的另一个表。调用 SqlTransaction 对象的 Commit 方法，提交事务。之后，就可以在数据库中看到新记录：

```
public static void TransactionSample()
{
    //...
    command.Parameters["Title"].Value = "Professional C# 9 and .NET Core 4.0";
    command.Parameters["Publisher"].Value = "Wrox Press";
    command.Parameters["Isbn"].Value = "42-08154711";
    command.Parameters["ReleaseDate"].Value = new DateTime(2022, 11, 2);

    id = await command.ExecuteScalarAsync();
    Console.WriteLine($"record added with id: {id}");

    tx.Commit();
  }
  catch (Exception ex)
  {
    Console.WriteLine($"error {ex.Message}, rolling back");
    tx.Rollback();
  }
}
```

检查了两个记录的 Isbn 号吗？它们是相同的。由于在 Isbn 号上使用唯一索引指定了数据库表，因此写入第二个记录会失败，抛出类型 SqlException 的异常，异常信息是 Cannot insert duplicate key row in object 'ProCSharp.Books' with unique index 'IX_Books_Isbn'. The duplicate key value is (42-8154711)，因此，Rollback 方法会撤销同一事务中的所有 SQL 命令。状态重置为事务启动之前的状态。这样，第一个记录也不会写入数据库。

如果在调试模式下运行程序，断点激活的时间太长，事务就会中断，因为事务超时了。事务处于活跃状态时，并不意味着有用户输入。为用户输入增加事务的超时时间也不是很有用，因为事务处于活跃状态，会导致在数据库中有一个锁定。根据读写的记录，可能出现行锁、页锁或表锁。为创建事务设置隔离级别，可以影响锁定，因此影响数据库的性能。然而，这也影响事务的 ACID 属性，例如，并不是所有数据都是隔离的。

应用于事务的默认隔离级别是 ReadCommitted。表 25-1 显示了可以设置的不同选项。

表 25-1

| 隔 离 级 别 | 说　明 |
| --- | --- |
| ReadUncommitted | 使用 ReadUncommitted，事务不会相互隔离。使用这个级别，不等待其他事务释放锁定的记录。这样，就可以从其他事务中读取未提交的数据——脏读。这个级别通常仅用于读取不管是否读取临时修改都无关紧要的记录，如报表 |
| ReadCommitted | ReadCommitted 等待其他事务释放对记录的写入锁定。这样，就不会出现脏读操作。这个级别为读取当前的记录设置读取锁定，为要写入的记录设置写入锁定，直到事务完成为止。对于要读取的一系列记录，在移动到下一个记录上时，前一个记录都是未锁定的，所以可能出现不可重复的读操作 |
| RepeatableRead | RepeatableRead 为读取的记录设置锁定，直到事务完成为止。这样，就避免了不可重复读的问题。但幻读仍可能发生 |
| Serializable | Serializable 设置范围锁定。在运行事务时，不可能添加与所读取的数据位于同一个范围的新记录 |
| Snapshot | Snapshot 用于对实际的数据建立快照。在复制修改的记录时，这个级别会减少锁定。这样，其他事务仍可以读取旧数据，而不需要等待解锁 |
| Unspecified | Unspecified 表示，提供程序使用另一个隔离级别值，该值不同于 IsolationLevel 枚举定义的值 |
| Chaos | Chaos 类似于 ReadUncommitted，但除了执行 ReadUncommitted 值的操作之外，它不能锁定更新的记录 |

表 25-2 总结了设置最常用的事务隔离级别可能导致的问题。

表 25-2

| 隔 离 级 别 | 脏 读 | 不可重复读 | 幻 读 |
| --- | --- | --- | --- |
| ReadUncommitted | Y | Y | Y |
| ReadCommitted | N | Y | Y |
| RepeatableRead | N | N | Y |
| Serializable | Y | Y | Y |

## 25.6 事务和 System.Transaction

处理事务的另一种方法是使用 System.Transactions。这些类自从.NET Framework 2.0 开始就可以使用，但是在.NET Core 1.1 中还不可用。在.NET Core 2.1 中，这些类型又回来了，可以从 ADO.NET(从 System.Data.SqlClient 的 4.5 版本开始)和 Entity Framework Core 2.1 中使用。

System.Transactions 名称空间为事务定义了几个类。可能最重要的类是 Transaction。它可以用于直接访问环境事务，提供关于事务的信息，以及在发生故障时启动回滚。表 25-3 描述了 Transaction 类的成员：

表 25-3

| Transaction 类的成员 | 说 明 |
| --- | --- |
| Current | Current 是一个静态属性。如果存在环境事务，则 Transaction.Current 返回该事务。本章后面将讨论环境事务 |
| IsolationLevel | IsolationLevel 属性返回一个 IsolationLevel 类型的对象。IsolationLevel 是一个枚举，它定义了其他事务对某事务的临时结果的访问权限。隔离级别的信息可参阅使用本章的 25.5 节 |
| TransactionInformation | TransactionInformation 属性返回 TransactionInformation 对象，它提供关于事务当前状态、创建事务的时间和事务标识符的信息 |
| Rollback | 使用 Rollback 方法，可以中止事务，并撤销事务的所有部分，将所有结果设置为事务之前的状态 |
| DependentClone | 使用 DependentClone 方法，可以创建一个依赖当前事务的事务 |
| TransactionCompleted | TransactionCompleted 是在事务完成时触发的事件——要么成功，要么失败。使用 TransactionCompletedEventHandler 类型的事件处理程序对象，可以访问 Transaction 对象并读取其状态。 |

示例应用程序(.NET Core Console App)显示，System.Transactions 系统的特性使用了以下依赖项和名称空间：

**依赖项**

Microsoft.Extensions.Configuration
Microsoft.Extensions.Configuration.Json
System.Data.SqlClient

**名称空间**

Microsoft.Extensions.Configuration
System
System.Data.SqlClient
System.IO
System.Linq
System.Threading.Tasks
System.Transactions

代码示例定义了一个 Utilities 类，其中的一些辅助方法在不同示例中使用。(代码文件 SystemTransactionSamples/Utilities.cs)：

```
public class Utilities
{
  public static bool AbortTx()
  {
    Console.WriteLine("Abort the transaction (y/n)?");
    return Console.ReadLine().ToLower().Equals("y");
  }

  public static void DisplayTransactionInformation(string title,
    TransactionInformation info)
  {
    if (info == null) throw new ArgumentNullException(nameof(info));

    Console.WriteLine(title);
    Console.WriteLine($"Creation time: {info.CreationTime:T}");
    Console.WriteLine($"Status: {info.Status}");
    Console.WriteLine($"Local Id: {info.LocalIdentifier}");
    Console.WriteLine($"Distributed Id: {info.DistributedIdentifier}");
    Console.WriteLine();
  }
}
```

**注意：**

示例代码询问用户是否应该终止事务。这适用于演示，但是在实际应用程序的事务中不应该有用户交互。当事务处于活动状态时，锁定记录。我们不希望妨碍其他用户工作，原因是一个用户因为这些锁打开了锁。

还需要注意事务超时。如果用户输入占用的时间超过事务超时时间，事务将被中止。

这个示例使用与前面示例相同的数据库，但是有一个重要的更改。这次 Book 类型定义为从 Books 表中映射信息(代码文件 SystemTransactionSamples/Book.cs)：

```
public class Book
{
  public int Id { get; set; }
  public string Title { get; set; }
  public string Publisher { get; set; }
  public string Isbn { get; set; }
  public DateTime? ReleaseDate { get; set; }
}
```

BookData 类将 Book 类型映射到数据库，并实现了 AddBookAsync 方法，以向 Books 表添加新记录。该实现与以前在调用 ExecuteNonQuery 方法插入新记录时看到的实现类似，但有一个重要的区别。这次 System.Transactions.Transaction 使用 SqlConnection 方法 EnlistTransaction 征募。这样，SqlConnection 将参与该事务的结果(代码文件 SystemTransactionSamples/BookData.cs)：

```
public class BookData
{
  public async Task AddBookAsync(Book book, Transaction tx)
  {
    using (SqlConnection connection = new SqlConnection(GetConnectionString()))
    {
      string sql = "INSERT INTO [ProCSharp].[Books] ([Title], [Publisher], " +
        "[Isbn], [ReleaseDate]) " +
        "VALUES (@Title, @Publisher, @Isbn, @ReleaseDate)";

      await connection.OpenAsync();
      if (tx != null)
      {
        connection.EnlistTransaction(tx);
      }
      var command = connection.CreateCommand();
      command.CommandText = sql;
      command.Parameters.AddWithValue("Title", book.Title);
      command.Parameters.AddWithValue("Publisher", book.Publisher);
      command.Parameters.AddWithValue("Isbn", book.Isbn);
      command.Parameters.AddWithValue("ReleaseDate", book.ReleaseDate);

      await command.ExecuteNonQueryAsync();
    }
  }
  //...
}
```

### 25.6.1　可提交的事务

Transaction 类不能以编程方式提交；它没有提交事务的方法。基类 Transaction 只支持中止事务。唯一支持提交事务的类是类 CommittableTransaction。

在 CommittableTransactionAsync 方法中，首先创建一个类型为 CommittableTransaction 的事务，并向控制台显示信息。然后创建一个 Book 对象，该对象在 AddBookAsync 方法中写入数据库。如果在事务外部验证数据库中的记录，则在事务完成之前无法看到添加的图书。如果事务失败，就会出现回滚，并且不会将图书写到数据库中。

在调用 AddBookAsync 方法之后，调用 AbortTx 方法来询问用户是否应该中止事务。如果用户中止，则抛出 ApplicationException 类型的异常，在 catch 块中，通过调用 Transaction 类的 Rollback 方法执行回滚。记录不会写入数据库。如果用户没有中止，Commit 方法将提交事务，并提交事务的最终状态(代码文件 SystemTransactionSamples/Program.cs)：

```
static async Task CommittableTransactionAsync()
{
  var tx = new CommittableTransaction();
  DisplayTransactionInformation("TX created", tx.TransactionInformation);

  try
  {
    var b = new Book
    {
      Title = "A Dog in The House",
      Publisher = "Pet Show",
      Isbn = RandomIsbn(),
      ReleaseDate = new DateTime(2018, 11, 24)
    };
    var data = new BookData();
    await data.AddBookAsync(b, tx);

    if (AbortTx())
    {
      throw new ApplicationException("transaction abort by the user");
    }
    tx.Commit();
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
    Console.WriteLine();
    tx.Rollback();
  }

  DisplayTransactionInformation("TX completed", tx.TransactionInformation);
}
```

如下面的应用程序输出所示，事务是活动的，并且具有一个本地标识符。此外，用户选择中止事务。事务完成后，可以看到中止状态：

```
TX created
Creation time: 7:29:36 PM
Status: Active
Local Id: c090c903-3f74-44b2-92a7-7607e33b787c:1
Distributed Id: 00000000-0000-0000-0000-000000000000

Abort the transaction (y/n)?
y
transaction abort by the user

TX completed
Creation time: 7:29:36 PM
Status: Aborted
Local Id: c090c903-3f74-44b2-92a7-7607e33b787c:1
Distributed Id: 00000000-0000-0000-0000-000000000000
```

接下来是应用程序的第二个输出，用户不会中止事务。事务的状态是 Committed，数据写入数据库：

```
TX created
Creation time: 7:30:59 PM
```

```
Status: Active
Local Id: e60feb38-e3b2-4ede-992d-f93296a363a4:1
Distributed Id: 00000000-0000-0000-0000-000000000000

Abort the transaction (y/n)?
n
TX completed
Creation time: 7:30:59 PM
Status: Committed
Local Id: e60feb38-e3b2-4ede-992d-f93296a363a4:1
Distributed Id: 00000000-0000-0000-0000-000000000000
```

### 25.6.2 依赖事务

对于依赖事务，可以在多个任务或线程之间影响一个事务。依赖事务依赖于另一个事务，并影响事务的结果。

示例方法 DependentTransactions 首先使用 CommittableTransaction 创建根事务。此事务对象使用方法 DependentClone 创建一个依赖事务。该 DependentTransaction 在本地函数 UsingDependentTransactionAsync 中传递给一个单独的任务。依赖事务可以标记为已完成，就像调用 Complete 方法一样。

DependentClone 方法需要 DependentCloneOption 类型的参数，该参数是一个枚举，其值为 BlockCompleteUntilComplete 和 RollbackIfNotComplete。如果根事务在依赖事务之前完成，则此选项非常重要。将该选项设置为 RollbackIfNotComplete，如果依赖事务没有在根事务的 Commit 方法之前调用 Complete 方法，则事务将中止。将该选项设置为 BlockCommitUntilComplete，方法 Commit 就会等待(阻塞)，直到所有依赖事务都定义了结果为止。

接下来，如果用户不中止事务，则调用 CommittableTransaction 类的 Commit 方法(代码文件 SystemTransactions/Program.cs)：

```
static void DependentTransactions()
{
  async Task UsingDependentTransactionAsync(DependentTransaction dtx)
  {
    dtx.TransactionCompleted += (sender, e) =>
      DisplayTransactionInformation("Depdendent TX completed",
        e.Transaction.TransactionInformation);

    DisplayTransactionInformation("Dependent Tx", dtx.TransactionInformation);

    await Task.Delay(2000);

    dtx.Complete();
    DisplayTransactionInformation("Dependent Tx send complete",
      dtx.TransactionInformation);
  }

  var tx = new CommittableTransaction();
  DisplayTransactionInformation("Root Tx created",
    tx.TransactionInformation);

  try
  {
    DependentTransaction depTx = tx.DependentClone(
      DependentCloneOption.BlockCommitUntilComplete);
    Task t1 = Task.Run(() => UsingDependentTransactionAsync(depTx));

    if (AbortTx())
    {
      throw new ApplicationException("transaction abort by the user");
    }
    tx.Commit();
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
    tx.Rollback();
  }

  DisplayTransactionInformation("TX completed", tx.TransactionInformation);
}
```

应用程序的以下输出显示了根事务及其标识符。因为选项 DependentCloneOption.BlockCommitUntilComplete，根事务在 Commit 方法中等待，直到定义了依赖事务的结果为止。一旦依赖事务完成，就提交事务：

```
Root Tx created
Creation time: 7:50:32 PM
Status: Active
Local Id: 87268b8c-076d-4823-af58-944b861bd4fe:1
Distributed Id: 00000000-0000-0000-0000-000000000000

Abort the transaction (y/n)?
Dependent Tx
Creation time: 7:50:32 PM
Status: Active
Local Id: 87268b8c-076d-4823-af58-944b861bd4fe:1
Distributed Id: 00000000-0000-0000-0000-000000000000

Dependent Tx send complete
Creation time: 7:50:32 PM
Status: Active
Local Id: 87268b8c-076d-4823-af58-944b861bd4fe:1
Distributed Id: 00000000-0000-0000-0000-000000000000

n
Depdendent TX completed
Creation time: 7:50:32 PM
Status: Committed
Local Id: 87268b8c-076d-4823-af58-944b861bd4fe:1
Distributed Id: 00000000-0000-0000-0000-000000000000

TX completed
Creation time: 7:50:32 PM
Status: Committed
Local Id: 87268b8c-076d-4823-af58-944b861bd4fe:1
Distributed Id: 00000000-0000-0000-0000-000000000000
```

### 25.6.3　环境事务

System.Transactions 名称空间中类的最大优势是环境事务特性。对于环境事务，不需要手动获取与事务的连接；这是由支持环境事务的资源自动完成的。

环境事务与当前线程关联。可以使用静态属性 Transaction.Current 获取并设置环境事务。支持环境事务的 API 检查此属性，以获得环境事务，并与事务合并。ADO.NET 连接支持环境事务。

可以创建一个 CommittableTransaction 对象并将其分配给 Transaction.Current 属性，来初始化环境事务。另一种方法(通常是首选方法)是使用 TransactionScope 类。TransactionScope 的构造函数会创建一个环境事务。

**注意：**
*如果 ADO.NET 连接不应该与环境事务合并，则可以用连接字符串设置值 Enlist= false。*

为了使用 TransactionScope，创建了另一个 AddBook 方法，它只向 Books 表添加一条记录，而不显式地向事务注册(代码文件 SystemTransactionSamples/BookData.cs)：

```
public void AddBook(Book book)
{
  using (SqlConnection connection = new SqlConnection(GetConnectionString()))
  {
    string sql = "INSERT INTO [ProCSharp].[Books] ([Title], [Publisher], " +
      "[Isbn], [ReleaseDate]) " +
      "VALUES (@Title, @Publisher, @Isbn, @ReleaseDate)";

    connection.Open();

    var command = connection.CreateCommand();
    command.CommandText = sql;
```

```
      command.Parameters.AddWithValue("Title", book.Title);
      command.Parameters.AddWithValue("Publisher", book.Publisher);
      command.Parameters.AddWithValue("Isbn", book.Isbn);
      command.Parameters.AddWithValue("ReleaseDate", book.ReleaseDate);

      command.ExecuteNonQuery();
    }
  }
```

TransactionScope 的重要方法是 Complete 和 Dispose。Complete 方法为作用域设置成功位，如果作用域是根作用域，那么 Dispose 方法将结束作用域，并提交或回滚事务。

因为 TransactionScope 类实现了 IDisposable 接口，所以可以使用 using 语句定义作用域。默认构造函数创建一个新事务。下面的代码片段在创建 TransactionScope 实例之后，立即将调用 DisplayTransactionInformation 的 lambda 表达式分配给 TransactionCompleted 事件，以便在事务完成时获取信息。在此事件触发之前，在创建事务之后，访问 Transaction.Current 属性，以在控制台上显示事务的当前状态。然后，创建一个新的 Book 对象，并调用先前创建的 AddBook 方法。如果用户没有中止事务，则调用 TransactionScope 的 Complete 方法。

using 块的末尾调用 TransactionScope 的 Dispose 方法。如果调用了 Complete 方法，并且参与事务的所有其他方(例如，SQL 连接)都设置了成功位，则提交事务。如果任何一方未成功处理包含 TransactionScope 的事务，也未调用 Complete，则中止事务(代码文件 SystemTransactionSamples/Program.cs):

```
static void AmbientTransactions()
{
  using (var scope = new TransactionScope())
  {
    Transaction.Current.TransactionCompleted += (sender, e) =>
      DisplayTransactionInformation("TX completed",
        e.Transaction.TransactionInformation);

    DisplayTransactionInformation("Ambient TX created",
      Transaction.Current.TransactionInformation);

    var b = new Book
    {
      Title = "Cats in The House",
      Publisher = "Pet Show",
      Isbn = RandomIsbn(),
      ReleaseDate = new DateTime(2019, 11, 24)
    };
    var data = new BookData();
    data.AddBook(b);

    if (!AbortTx())
    {
      scope.Complete();
    }
    else
    {
      Console.WriteLine("transaction abort by the user");
    }

  } // scope.Dispose();
}
```

运行应用程序时，可以在创建 TransactionSope 类的实例之后看到一个活动的环境事务。应用程序的最后一个输出是 TransactionComplete 事件处理程序的输出，以显示已完成的事务状态:

```
Ambient TX created
Creation time: 7:53:10 AM
Status: Active
Local Id: 0e28b708-9dd7-4b17-bf41-c09f963928cf:1
Distributed Id: 00000000-0000-0000-0000-000000000000

Abort the transaction (y/n)?
n
TX completed
Creation time: 7:53:10 AM
Status: Committed
Local Id: 0e28b708-9dd7-4b17-bf41-c09f963928cf:1
Distributed Id: 00000000-0000-0000-0000-000000000000
```

### 25.6.4　嵌套作用域和环境事务

对于 TransactionScope 类，还可以嵌套作用域。嵌套的作用域可以直接位于外部作用域内，也可以位于在作用域中调用的方法内。嵌套的作用域可以使用与外部作用域相同的事务，抑制事务，或者创建独立于外部作用域的新事务。对该作用域的需求由 TransactionScopeOption 枚举定义，该枚举传递给 TransactionScope 类的构造函数。

表 25-4 描述了 TransactionScope 枚举可用的值和相应功能。

表 25-4

| TransactionScope 成员 | 说　明 |
|---|---|
| Required | Required 指定，作用域需要一个事务。如果外部作用域已经包含一个环境事务，内部作用域就使用现有事务。如果环境事务不存在，则创建一个新的事务。如果两个作用域共享相同的事务，那么每个作用域都会影响事务的结果。只有当所有作用域设置成功位时，事务才能提交。只要有一个作用域在根作用域被释放之前没有调用 Complete 方法，事务就中止 |
| RequiresNew | RequiresNew 总是创建一个新的事务。如果外部作用域已经定义了事务，那么来自内部作用域的事务是完全独立的。两个事务都可以单独提交或中止 |
| Suppress | 使用 Suppress，无论外部作用域是否包含事务，作用域都不包含环境事务 |

下一个示例定义了两个作用域。使用选项 TransactionScopeOption.RequiresNew，内部作用域配置为需要一个新事务（代码文件 SystemTransactionSamples/Program.cs）：

```
static void NestedScopes()
{
  using (var scope = new TransactionScope())
  {
    Transaction.Current.TransactionCompleted += (sender, e) =>
      DisplayTransactionInformation("TX completed",
        e.Transaction.TransactionInformation);

    DisplayTransactionInformation("Ambient TX created",
      Transaction.Current.TransactionInformation);

    var b = new Book
    {
      Title = "Dogs in The House",
      Publisher = "Pet Show",
      Isbn = RandomIsbn(),
      ReleaseDate = new DateTime(2020, 11, 24)
    };
    var data = new BookData();
    data.AddBook(b);

    using (var scope2 = new
      TransactionScope(TransactionScopeOption.RequiresNew))
    {
      Transaction.Current.TransactionCompleted += (sender, e) =>
        DisplayTransactionInformation("Inner TX completed",
          e.Transaction.TransactionInformation);

      DisplayTransactionInformation("Inner TX scope",
        Transaction.Current.TransactionInformation);

      var b1 = new Book
      {
        Title = "Dogs and Cats in The House",
        Publisher = "Pet Show",
        Isbn = RandomIsbn(),
        ReleaseDate = new DateTime(2021, 11, 24)
      };
      var data1 = new BookData();
      data1.AddBook(b1);

      scope2.Complete();
```

```
    }

    scope.Complete();
  }
```

在运行应用程序时，可以看到外部作用域和内部作用域的不同事务标识符，其中内部作用域在 GUID 上附加:2，外部作用域在 GUID 上附加:1。这些事务彼此独立：

```
Ambient TX created
Creation time: 8:20:09 AM
Status: Active
Local Id: d4e3a180-49d6-4c16-b4c9-89a9c6d4ace9:1
Distributed Id: 00000000-0000-0000-0000-000000000000

Inner TX scope
Creation time: 8:20:11 AM
Status: Active
Local Id: d4e3a180-49d6-4c16-b4c9-89a9c6d4ace9:2
Distributed Id: 00000000-0000-0000-0000-000000000000

Inner TX completed
Creation time: 8:20:11 AM
Status: Committed
Local Id: d4e3a180-49d6-4c16-b4c9-89a9c6d4ace9:2
Distributed Id: 00000000-0000-0000-0000-000000000000

TX completed
Creation time: 8:20:09 AM
Status: Committed
Local Id: d4e3a180-49d6-4c16-b4c9-89a9c6d4ace9:1
Distributed Id: 00000000-0000-0000-0000-000000000000
```

如果将内部作用域的创建更改为使用 TransactionScopeOption.Required，就可以看到同样的事务用于外部和内部作用域：

```
Ambient TX created
Creation time: 8:23:52 AM
Status: Active
Local Id: 95181f71-0268-40f0-8f12-471a01a83638:1
Distributed Id: 00000000-0000-0000-0000-000000000000

Inner TX scope
Creation time: 8:23:52 AM
Status: Active
Local Id: 95181f71-0268-40f0-8f12-471a01a83638:1
Distributed Id: 00000000-0000-0000-0000-000000000000

TX completed
Creation time: 8:23:52 AM
Status: Committed
Local Id: 95181f71-0268-40f0-8f12-471a01a83638:1
Distributed Id: 00000000-0000-0000-0000-000000000000

Inner TX completed
Creation time: 8:23:52 AM
Status: Committed
Local Id: 95181f71-0268-40f0-8f12-471a01a83638:1
Distributed Id: 00000000-0000-0000-0000-000000000000
```

## 25.7 小结

本章介绍了 ADO.NET 的核心基础。首先介绍的 SqlConnection 对象打开一个到 SQL Server 的连接。讨论了如何从配置文件中检索连接字符串。

接着阐述了如何正确地进行连接，这样稀缺的资源就可以尽可能早地关闭。所有连接类都实现 IDisposable 接口，在对象放在 using 子句中时调用该接口。如果本章只有一件值得注意的事，那就是尽早关闭数据库连接的重要性。

对于命令，传递参数，就得到一个返回值，使用 SqlDataReader 检索记录。还论述了如何使用 SqlCommand 对象调用存储过程。

类似于框架的其他部分，处理可能要花一些时间，ADO.NET 实现了基于任务的异步模式。还看到了如何通过 ADO.NET 创建和使用事务。

对于 System.Transactions，介绍了处理事务的另一种方法——可以独立于 SQL 连接的事务，而在使用环境事务时，SQL 连接会自动将其列到事务中。

第 26 章讨论 ADO.NET Entity Framework Core，它提供了关系数据库和对象层次结构之间的映射，从而提供了抽象的数据访问。访问关系数据库时，在后台使用 ADO.NET 类。

# 第 26 章

# Entity Framework Core

**本章要点**

- Entity Framework Core 简介
- 使用依赖项注入和 Entity Framework
- 约定、注释和流利 API
- 使用查询、已编译的查询和全局查询过滤器
- 通过约定、注释和流利 API 定义关系
- 在每个层次结构中使用表、表分割和拥有的实体
- 对象跟踪
- 更新对象和对象树
- 用更新处理冲突
- 使用事务
- 使用迁移和.NET CLI 工具

**本章源代码下载地址(wrox.com)：**

打开 www.wrox.com 的 Download Code 选项卡可下载本章源代码。源代码也可以在 EFCore 目录的 https://github.com/ProfessionalCSharp/ProfessionalCSharp7 中找到。本章代码分为以下几个主要的示例文件：

- Intro
- Books Sample
- UsingDependencyInjection
- Menus Sample
- Menus with Data Annotations
- Relations with Annotations/Conventions/FluentAPI
- Table per Hierarchy
- Conflict Handling Sample
- Transactions Sample

## 26.1　Entity Framework 简史

Entity Framework 和 Entity Framework Core (EF Core)是一个提供了实体-关系映射的架构，通过它们，可以创建映射到数据库表的类型，使用 LINQ 创建数据库查询，创建和更新对象，把它们写入数据库。

Entity Framework 经过多年的改变，EF Core 完全重写了。下面看看 Entity Framework 的历史，了解改写的原因。

- Entity Framework 1——Entity Framework 的第一个版本没有准备用于.NET 3.5，但不久它就可用于.NET 3.5 SP1。另一个产品是 LINQ to SQL，它提供了类似的功能，可用于.NET 3.5。从广义上看，LINQ to SQL 和 Entity Framework 提供了类似的功能。然而，LINQ to SQL 使用起来更简单，但只用于访问 SQL Server。Entity Framework 是基于提供程序的，可以访问几种不同的关系数据库。它包含了更多的功能，比如多对多映射，不需要映射对象，可以进行 n 到 n 映射。Entity Framework 的一个缺点是，它要求模型类型派生自 EntityObject 基类。使用一个包含 XML 的 EDMX 文件，把对象映射到关系上。所包含的 XML 用三种模式定义：概念模式定义(Conceptual Schema Definition，CSD)定义对象类型及其属性和关联；存储模式定义(Storage Schema Definition，SSD)定义了数据库表、列和关系；映射模式语言(Mapping Schema Language，MSL)定义了 CSD 和 SSD 如何彼此映射。
- Entity Framework 4——Entity Framework 4 可用于.NET 4，进行了重大改进，许多想法都来自 LINQ to SQL。因为改动较大，跳过了版本 2 和 3。在这个版本中，增加了延迟加载，在访问属性时获取关系。设计模型后，可以使用 SQL 数据定义语言(DDL)创建数据库。使用 Entity Framework 的两个模型现在是 Database First 或 Model First。添加的最重要特性是支持 Plain Old CLR Objects (POCO)，所以不再需要派生自基类 EntityObject。

在后来的更新(如 Entity Framework 4.1、4.2)中，用 NuGet 包添加了额外的特性。这允许更快地增加功能。Entity Framework 4.1 提供了 Code First 模型，其中不再使用定义映射的 EDMX 文件。相反，所有的映射都使用 C#代码定义——使用特性或流利 API 定义使用代码的映射。

Entity Framework 4.3 添加了对迁移的支持。有了迁移，可以使用 C#代码定义对数据库中模式的更新。数据库更新可以自动应用到使用数据库的应用程序上。

- Entity Framework 5——Entity Framework 5 的 NuGet 包支持.NET Framework 4.5 和.NET Framework 4.0 应用程序。然而， Entity Framework 5 的许多功能可用于.NET Framework 4.5。Entity Framework 仍然基于安装在系统上的类型和.NET Framework 4.5。在这个版本中，新增了性能改进，支持新的 SQL Server 功能，如空间数据类型。而且，当使用.NET Framework 4.0 时，这些特性中有很多都是无法使用的。
- Entity Framework 6——Entity Framework 6 解决了 Entity Framework 5 的一些问题，其部分原因是，该框架的一部分安装在系统上，一部分通过 NuGet 扩展获得。现在，Entity Framework 的完整代码都移动到 NuGet 包上。为了不出现冲突，使用了一个新的名称空间。将应用程序移植到新版本上，必须改变名称空间。
- Entity Framework Core (EF Core)——Entity Framework 的这个版本有了新的名称，是对 Entity Framework 的完全重写。EF Core 不仅可以在 Windows 上使用，也可以在 Linux 和 Mac 上使用。它支持关系数据库和 NoSQL 数据存储。

本书介绍 Entity Framework Core 2.0。这个版本不支持 XML 文件映射与 CSDL、SSDL 和 MSL。只支持 Code First—— 用 Entity Framework 4.1 添加的模型。Code First 并不意味着数据库不存在。可以先创建数据库，或纯粹从代码中定义数据库；这两种选择都是可能的。

**注意：**

名称 Code First 有些误导。在 Code First 中，代码或者数据库都可以先创建。在最初 Code First 的 beta 版本中，名字是 Code Only。因为其他模型选项在名字中包含 First，所以名称 Code Only 也改变了。

Entity Framework Core 1.0 不支持 Entity Framework 6 提供的所有特性。Entity Framework Core 2.0 有所改进，但仍然不支持 Entity Framework 6 的所有特性。然而，它也有一些 Entity Framework 6 没有的新特性。

只需要注意使用什么版本的 Entity Framework。始终使用 Entity Framework 6 有许多有效的理由，但在非 Windows 平台上使用 ASP.NET Core，使用 Entity Framework 与通用 Windows 平台，使用 Xamarin，使用非关系数据存储，都需要使用 EF Core。

本章介绍 EF Core。它始于一个简单的模型读写来自 SQL Server 的信息。后来，添加了关系，在写入数据库时介绍变更追踪器和冲突的处理。使用迁移创建和修改数据库模式是本章的另一个重要组成部分。

**注意：**

本章使用 Books 数据库。这个数据库包含在 www.wrox.com 下载的代码示例中。与示例一起使用的其他数据库是从代码中创建的。

## 26.2 EF Core 简介

第一个例子使用了一个 Book 类型，把这种类型映射到 SQL Server 数据库中的 Books 表。把记录写到数据库，然后读取、更新和删除它们。

在第一个示例中，首先创建数据库或者从应用程序创建数据库。为了先创建数据库，可以使用 Visual Studio 2017 中的 SQL Server Object Explorer。选择数据库实例(localdb)\ MSSQLLocalDB(随 Visual Studio 一起安装)，单击树视图中的 Databases 节点，然后选择 Add New Database。示例数据库 WroxBooks 只有一个表 Books。

为了创建 Books 表，可以在 WroxBooks 数据库中选择 Tables 节点，然后选择 Add New Table。使用如图 26-1 所示的设计器，或者在 T-SQL 编辑器中输入 SQL DDL 语句，就可以创建 Books 表。下面的代码片段显示了创建表的 T-SQL 代码。单击 Update 按钮，可以将更改提交到数据库。

```
CREATE TABLE [dbo].[Books]
(
  [BookId] INT IDENTITY(1, 1) NOT NULL,
  [Title] NVARCHAR(50) NOT NULL,
  [Publisher] NVARCHAR(25) NULL,
  CONSTRAINT [PK_Books] PRIMARY KEY CLUSTERED ([BookId] ASC)
)
```

本章使用的示例应用程序都是.NET Core 控制台应用程序，使用以下依赖项和名称空间：

**依赖项**

Microsoft.EntityFrameworkCore

Microsoft.EntityFrameworkCore.SqlServer

Microsoft.EntityFramework.Design

Microsoft.Extensions.DependencyInjection

Microsoft.Extensions.Logging.Console

**名称空间**

Microsoft.EntityFrameworkCore

Microsoft.EntityFrameworkCore.ChangeTracking

Microsoft.EntityFrameworkCore.Diagnostics

Microsoft.EntityFrameworkCore.Infrastructure

Microsoft.EntityFrameworkCore.Metadata.Builders

Microsoft.Extensions.DependencyInjection

Microsoft.Extensions.Logging

System

System.Collections.Generic

System.ComponentModel.DataAnnotations

System.ComponentModel.DataAnnotations.Schema
System.Linq
System.Threading.Tasks

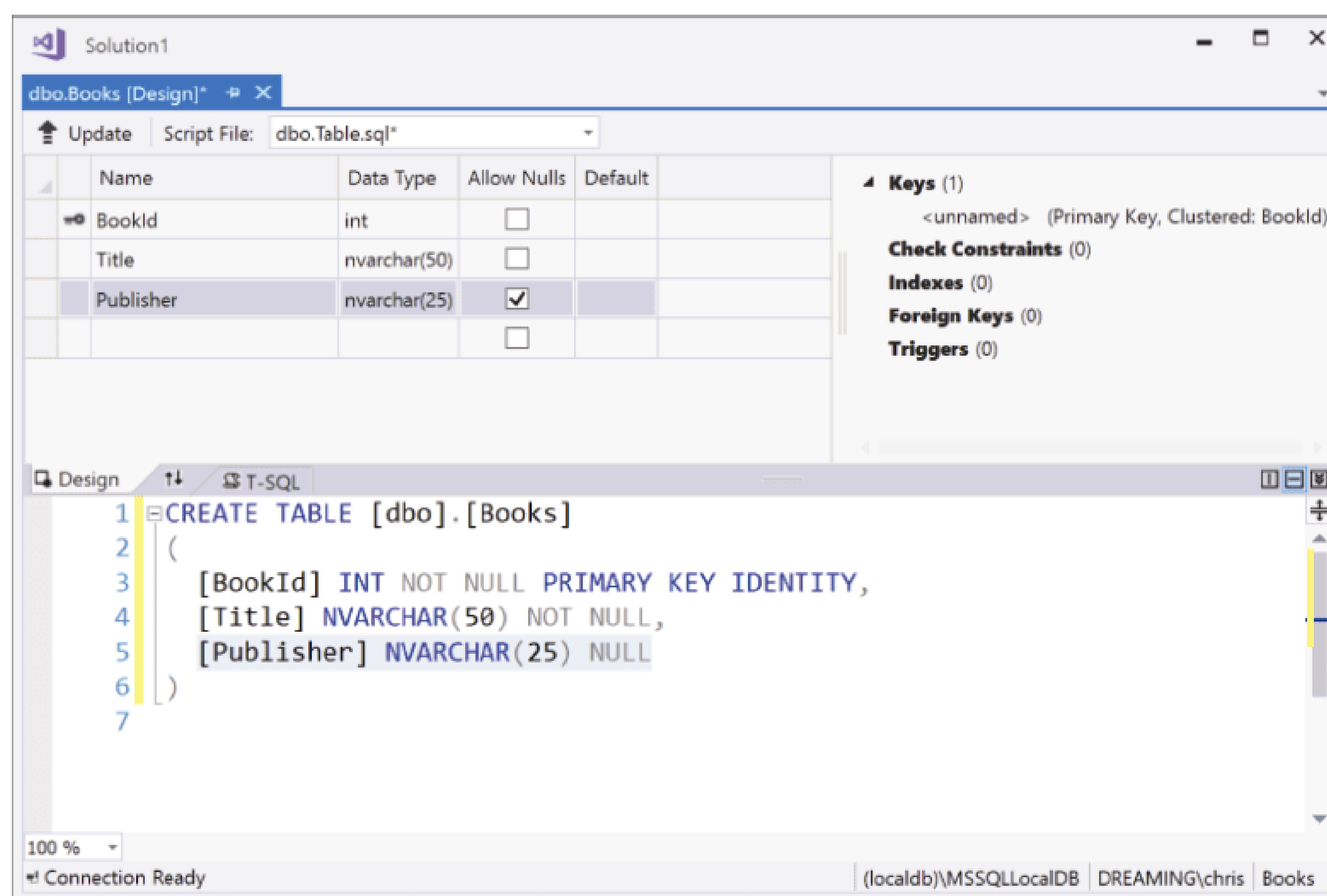



图 26-1

### 26.2.1　创建模型

访问 Books 数据库的 BookSample 示例应用程序是一个.NET Core 控制台应用程序。在这个应用程序中，Book 类是一个简单的实体类型，定义了三个属性。BookId 属性映射到表的主键，Title 属性映射到 Title 列，Publisher 属性映射到 Publisher 列。对于 Title 属性，应用 Required 属性是因为映射列在数据库中定义为 NOT NULL。使用 StringLength 属性应用 Title 和 Publisher 属性的长度。这也映射到数据库中的列。为了把类型映射到 Books 表，将 Table 特性应用于类型(代码文件 Intro/Book.cs)：

```
[Table("Books")]
public class Book
{
  public int BookId { get; set; }
  [Required]
  [StringLength(50)]
  public string Title { get; set; }
  [StringLength(30)]
  public string Publisher { get; set; }
}
```

### 26.2.2　约定、注释和流利 API

EF Core 使用了三个概念来定义模型：约定、注释和流利 API。按照约定，有些事情会自动发生。例如，用 Id 前缀命名 int 或 Guid 类型的属性，将该属性映射到主键。

可以使用注释重写约定——指定特性。前面的例子使用 Table 特性将 Book 类型映射到 Books 表。还有一个映射到表格的约定：使用上下文的属性名。下一节将展示如何创建上下文。并不是每个注释都有约定。还使用了 Required 和 StringLength 特性。注解比约定更强大；可以做得更多。

除了使用注释，还可以使用流利 API，这意味着配置是通过代码完成的，而不是使用特性完成的。在流利 API 中，可以使用方法的返回值来调用下一个方法。用于 EF Core 的流利 API 比注释更强大；可以做得更多。

### 26.2.3 创建上下文

通过创建 BooksContext 类，实现了 Book 表与数据库的关系。这个类派生自基类 DbContext。BooksContext 类定义了 DbSet<Book>类型的 Books 属性。这个类型允许创建查询，添加 Book 实例，存储在数据库中。要定义连接字符串，可以重写 DbContext 的 OnConfiguring 方法。在这里，UseSqlServer 扩展方法将上下文映射到 SQL Server 数据库(代码文件 BooksSample/BooksContext.cs)：

```
public class BooksContext: DbContext
{
  private const string ConnectionString =
    @"server=(localdb)\MSSQLLocalDb;database=WroxBooks;" +
    @"trusted_connection=true";

  public DbSet<Book> Books { get; set; }

  protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
  {
    base.OnConfiguring(optionsBuilder);
    optionsBuilder.UseSqlServer(ConnectionString);
  }
}
```

**注意：**
定义连接字符串的另一种选择是使用依赖注入，参见本章后面的内容。

### 26.2.4 创建数据库

前面定义了模型和上下文类。现在还可以以编程方式创建数据库。首先实例化 BooksContext 对象。using 语句确保在 using 作用域结束时关闭数据库连接。

使用 DbContext 的 Database 属性时，会返回一个 DatabaseFacade。可以使用它创建和删除数据库，并直接发送 SQL 语句。调用 EnsureCreatedAsync 方法，确保创建了数据库。如果数据库已经存在，此方法将返回 false。如果数据库不存在，则根据上下文和模型的定义创建数据库，并返回 true(代码文件 Intro/Program.cs)：

```
private async Task CreateTheDatabaseAsync()
{
  using (var context = new BooksContext())
  {
    bool created = await context.Database.EnsureCreatedAsync();
    string creationInfo = created ? "created" : "exists";
    Console.WriteLine($"database {creationInfo}");
  }
}
```

运行这个程序时，如果之前已经创建了这个数据库，那么字符串 database exists 就会写入控制台。如果之前没有创建数据库，就创建数据库，然后写入字符串 database created。

**注意：**
许多代码示例都使用了 EF Core 的异步方法，如 EnsureCreatedAsync 和 SaveChangesAsync。如果不需要异步功能(例如，在控制台应用程序或 Web 应用程序中)，则可以使用这些方法的同步变体。尽管异步有一些开销，但是这些 API 的同步版本会阻塞调用线程。EnsureCreated 和 SaveChanges 是同步 API，而 EnsureCreatedAsync 和 SaveChangesAsync 是异步 API。异步方法详见第 15 章和第 21 章。

**注意：**
使用上下文方法的异步变体允许在后台启动操作。但是，不能在同一上下文中并行地启动多个操作。在开始下一个操作之前，需要等待操作完成。

### 26.2.5 删除数据库

数据库的删除与它的创建非常类似。只需要调用 DatabaseFacade 的方法 EnsureDeletedAsync：

```
private async Task DeleteDatabaseAsync()
{
  Console.Write("Delete the database? ");
  string input = Console.ReadLine();
  if (input.ToLower() == "y")
  {
    using (var context = new BooksContext())
    {
      bool deleted = await context.Database.EnsureDeletedAsync();
      string deletionInfo = deleted ? "deleted" : "not deleted";
      Console.WriteLine($"database {deletionInfo}");
    }
  }
}
```

确保不删除不应该删除的数据库。注意所使用的连接字符串。

### 26.2.6 写入数据库

创建了数据库和 Books 表后，就可以用数据填充表了。创建 AddBookAsync 方法，把 Book 对象添加到数据库中。AddBookAsync 方法仅把 Book 对象添加到上下文中，不写入数据库。必须调用 SaveChangesAsync 方法把 Book 对象写入数据库(代码文件 Intro/Program.cs)：

```
private async Task AddBookAsync(string title, string publisher)
{
  using (var context = new BooksContext())
  {
    var book = new Book
    {
      Title = title,
      Publisher = publisher
    };
    await context.Books.AddAsync(book);
    int records = await context.SaveChangesAsync();
    Console.WriteLine($"{records} record added");
  }
  Console.WriteLine();
}
```

为了添加一组图书，可以使用 AddRange 方法：

```
private async Task AddBooksAsync()
{
  using (var context = new BooksContext())
  {
    var b1 = new Book
    {
      Title = "Professional C# 6 and .NET Core 1.0",
      Publisher = "Wrox Press"
    };
    var b2 = new Book
    {
      Title = "Professional C# 5 and .NET 4.5.1",
      Publisher = "Wrox Press"
    };
    var b3 = new Book
    {
      Title = "JavaScript for Kids",
      Publisher = "Wrox Press"
    };
    var b4 = new Book
    {
      Title = "Web Design with HTML and CSS",
      Publisher = "For Dummies"
    };
    await context.AddRangeAsync(b1, b2, b3, b4);
    int records = await context.SaveChangesAsync();
    Console.WriteLine($"{records} records added");
  }
```

```
  Console.WriteLine();
}
```

运行应用程序，调用这些方法，就可以使用 SQL Server Object Explorer 查看写入数据库的数据。

### 26.2.7 读取数据库

为了在 C#代码中读取数据，只需要调用 BooksContext，访问 Books 属性。访问该属性会创建一个 SQL 语句，从数据库中检索所有的书(代码文件 Intro/Program.cs)：

```
private async Task ReadBooksAsync()
{
  using (var context = new BooksContext())
  {
    List<Book> books = await context.Books.ToListAsync();
    foreach (var b in books)
    {
      Console.WriteLine($"{b.Title} {b.Publisher}");
    }
  }
  Console.WriteLine();
}
```

在调试期间打开 IntelliTrace Events 窗口，就可以看到发送到数据库的 SQL 语句(这需要 Visual Studio 企业版)：

```
SELECT [b].[BookId], [b].[Publisher], [b].[Title]
FROM [Books] AS [b]
```

Entity Framework 提供了一个 LINQ 提供程序。使用它可以创建 LINQ 查询来访问数据库。也可以使用方法语法，如下所示：

```
private async Task QueryBooksAsync()
{
  using (var context = new BooksContext())
  {
    List<Book> wroxBooks = context.Books
      .Where(b => b.Publisher == "Wrox Press")
      .ToListAsync();

    foreach (var b in wroxBooks)
    {
      Console.WriteLine($"{b.Title} {b.Publisher}");
    }
  }
  Console.WriteLine();
}
```

或使用声明性的 LINQ 查询语法：

```
var wroxBooks = await (from b in context.Books
                       where b.Publisher == "Wrox Press"
                       select b).ToListAsync();
```

使用两个语法变体，将这个 SQL 语句发送到数据库：

```
SELECT [b].[BookId], [b].[Publisher], [b].[Title]
FROM [Books] AS [b]
WHERE [b].[Publisher] = 'Wrox Press'
```

**注意：**
LINQ 参见第 12 章。

### 26.2.8 更新记录

更新记录很容易实现：修改用上下文加载的对象，并调用SaveChangesAsync(代码文件Intro/Program.cs)：

```
private async Task UpdateBookAsync()
{
  using (var context = new BooksContext())
  {
```

```
    int records = 0;
    Book book = await context.Books
      .Where(b => b.Title == "Professional C# 7")
      .FirstOrDefaultAsync();

    if (book != null)
    {
      book.Title = "Professional C# 7 and .NET Core 2.0";
      records = await context.SaveChangesAsync();
    }
    Console.WriteLine($"{records} record updated");
  }
  Console.WriteLine();
}
```

### 26.2.9　删除记录

最后，清理数据库，删除所有记录。为此，可以检索所有记录，并调用 Remove 或 RemoveRange 方法，把上下文中对象的状态设置为删除。现在调用 SaveChangesAsync 方法，从数据库中删除记录，并为每一个对象调用 SQL Delete 语句(代码文件 Intro/Program.cs)：

```
private async Task DeleteBooksAsync()
{
  using (var context = new BooksContext())
  {
    var books = context.Books;
    context.Books.RemoveRange(books);
    int records = await context.SaveChangesAsync();
    Console.WriteLine($"{records} records deleted");
  }
  Console.WriteLine();
}
```

**注意：**

*对象-关系映射工具，如 EF Core，并不适用于所有场景。使用示例代码删除所有对象不那么高效。使用单个 SQL 语句可以删除所有记录，而不是为每一条记录使用一个 DELETE 语句。具体操作参见第 25 章。EF Core 在这种场景中并没有那么糟糕，因为多个语句可以合并为一个批处理语句，如本章后面所述。*

了解了如何添加、查询、更新和删除记录，本章后面将介绍后台的功能，讨论使用 Entity Framework 的高级场景。

### 26.2.10　日志记录

为了查看发送到数据库的 SQL 语句，可以打开 SQL Server 的分析器，在 Visual Studio 中打开 Intellitrace Events (Debug | Windows | Intellitrace Events)，这需要 Visual Studio 的企业版，或者只是启用日志记录。使用日志记录，可以在自己喜欢的地方编写跟踪信息。

EF Core 在内部使用一个依赖注入容器(使用 Microsoft.Extensions.DependencyInjection)，它注册了接口 ILoggerFactory。可以访问这个接口，并注册自己的日志记录器提供程序。

下面的代码片段使用 BooksContext 注册一个新的日志记录器。首先，使用 GetInfrastructure 扩展方法检索上下文的 IServiceProvider。这个扩展方法是在名称空间 Microsoft.EntityFrameworkCore.Infrastructure 中定义的。使用 IServiceProvider，可以检索在容器中注册的服务，例如接口 ILoggerFactory。此接口用于在 EF Core 基础结构中编写日志信息。使用这个接口，可以添加日志提供程序，比如控制台日志提供程序。这个日志提供程序在 NuGet 包 Microsoft.Extensions.Logging.Console 中定义。该提供程序为 ILoggerFactory 定义了 AddConsole 扩展方法，以便于将其添加为日志提供程序。在这里，日志提供程序配置为编写信息日志(代码文件 Intro/Program.cs)：

```
private void AddLogging()
{
  using (var context = new BooksContext())
  {
    IServiceProvider provider = context.GetInfrastructure<IServiceProvider>();
    ILoggerFactory loggerFactory = provider.GetService<ILoggerFactory>();
```

```
        loggerFactory.AddConsole(LogLevel.Information);
      }
    }
```

需要在只有一个上下文的情况下进行此配置。注册是在 EF Core 的基础结构中完成的，因此一旦为应用程序配置了这个基础结构，日志记录就会在每个实例化的上下文中完成。例如，前面实现的 QueryBooksAsync 方法现在在控制台上显示了日志信息：

```
info: Microsoft.EntityFrameworkCore.Database.Command[200101]
      Executed DbCommand (131ms) [Parameters=[], CommandType='Text',
        CommandTimeout='30']
      SELECT [b].[BookId], [b].[Publisher], [b].[Title]
      FROM [Books] AS [b]
      WHERE [b].[Publisher] = N'Wrox Press'
```

**注意：**

依赖注入和 Microsoft.Extensions.DependencyInjection 详见第 20 章。关于日志和诊断的信息详见第 29 章。

## 26.3 使用依赖注入

EF Core 内置了对依赖注入的支持。使用 EF Core 和依赖注入容器也得到了有力的支持。它不是定义连接并利用 DbContext 派生类来使用 SQL Server，而是使用依赖注入框架来注入连接和 SQL Server 选项。

为了看到其操作，前面的示例用 BooksSampleWithDI 示例项目进行修改。

BooksContext 类现在看起来要简单许多，只是定义 Books 属性(代码文件 UsingDependencyInjection/BooksContext.cs)：

```
public class BooksContext : DbContext
{
  public BooksContext(DbContextOptions<BooksContext> options)
    : base(options) { }

  public DbSet<Book> Books { get; set; }
}
```

BooksService 是利用 BooksContext 的新类。在这里，BooksContext 通过构造函数注入功能来注入。方法 AddBooksAsync 和 ReadBooks 非常类似于前面的示例，但是它们使用 BooksService 类的上下文成员，而不是创建一个新的上下文(代码文件 UsingDependencyInjection/BooksService.cs)：

```
public class BooksService
{
  private readonly BooksContext _booksContext;
  public BooksService(BooksContext context) => _booksContext = context;

  public async Task AddBooksAsync()
  {
    var b1 = new Book
    {
      Title = "Professional C# 6 and .NET Core 1.0",
      Publisher = "Wrox Press"
    };
    var b2 = new Book
    {
      Title = "Professional C# 5.0 and .NET 4.5.1",
      Publisher = "Wrox Press"
    };
    var b3 = new Book
    {
      Title = "JavaScript for Kids",
      Publisher = "Wrox Press"
    };
    var b4 = new Book
    {
      Title = "Web Design with HTML and CSS",
      Publisher = "For Dummies"
    };

    _booksContext.AddRange(b1, b2, b3, b4);
    int records = await _booksContext.SaveChangesAsync();
```

```
    Console.WriteLine($"{records} records added");
  }

  public async Task ReadBooksAsync()
  {
    List<Book> books = await _booksContext.Books.ToListAsync();
    foreach (var b in books)
    {
      Console.WriteLine($"{b.Title} {b.Publisher}");
    }
    Console.WriteLine();
  }
}
```

依赖注入框架的容器在 InitializeServices 方法中初始化。这里创建了 ServiceCollection 实例，在这个集合中添加 BooksService 类，并进行短暂的生命周期管理。这样，每次请求这个服务时，就实例化 ServiceCollection。为了注册 Entity Framework 和 SQL Server，可以使用扩展方法 AddEntityFrameworkSqlServer 和 AddDbContext。AddDbContext 方法需要一个 Action 委托作为参数，来接收 DbContextOptionsBuilder 参数。有了这个选项参数，上下文可以使用 UseSqlServer 扩展方法来配置。这类似于前面示例中用 Entity Framework 注册 SQL Server 的功能(代码文件 UsingDependencyInjection/Program.cs)：

```
private void InitializeServices()
{
  const string ConnectionString =
    @"server=(localdb)\MSSQLLocalDb;database=Books;trusted_connection=true";
  var services = new ServiceCollection();
  services.AddTransient<BooksService>()
    .AddEntityFrameworkSqlServer()
    .AddDbContext<BooksContext>(options =>
      options.UseSqlServer(ConnectionString));
  //...

  Container = services.BuildServiceProvider();
}

public ServiceProvider Container { get; private set; }
```

服务的初始化以及使用 BooksService 在 Main()方法中完成。通过调用 IServiceProvider 的 GetService()方法检索 BooksService (代码文件 UsingDependencyInjection/Program.cs)：

```
static async Task Main()
{
  var p = new Program();
  p.InitializeServices();
  p.ConfigureLogging();
  var service = p.Container.GetService<BooksService>();
  await service.AddBooksAsync();
  service.ReadBooks();
}
```

运行应用程序时，可以看到，在 Books 数据库中添加和读取记录。

为了利用这个应用程序设置配置日志记录，可以通过 AddLogging 扩展方法把 ILoggerFactory 接口添加到 DI 容器中：

```
private void InitializeServices()
{
  const string ConnectionString =
    @"server=(localdb)\MSSQLLocalDb;database=Books;trusted_connection=true";
  var services = new ServiceCollection();
  services.AddTransient<BooksService>()
    .AddEntityFrameworkSqlServer()
    .AddDbContext<BooksContext>(options =>
      options.UseSqlServer(ConnectionString));
  services.AddLogging();

  Container = services.BuildServiceProvider();
}
```

接着配置日志记录。在 ConfigureLogging 方法的实现代码中，从 DI 容器中检索 ILoggerFactory。使用这个工厂，会添加控制台，以写入信息日志：

```
private void ConfigureLogging()
```

```
{
  ILoggerFactory loggerFactory = Container.GetService<ILoggerFactory>();
  loggerFactory.AddConsole(LogLevel.Information);
}
```

EF Core 本身通过注入服务来使用 ILoggerFactory 接口，因此 EF Core 日志现在写入控制台，如上例所示。

# 26.4 创建模型

本章的第一个示例映射到一个表。第二个更复杂的示例展示了如何创建表之间的关系。新数据库是通过编程创建的，当然也可以先创建数据库，再使用代码访问已有的数据库。

## 26.4.1 创建关系

下面开始创建模型。示例项目使用 MenuCard 和 Menu 类型定义了一对多关系。MenuCard 包含 Menu 对象的列表。这个关系由 List<Menu>类型的 Menu 属性定义(代码文件 MenusSample/MenuCard.cs)：

```
public class MenuCard
{
  public int MenuCardId { get; set; }
  public string Title { get; set; }
  public List<Menu> Menus { get; } = new List<Menu>();
  public override string ToString() => Title;
}
```

也可以在另一个方向上访问关系，Menu 可以使用 MenuCard 属性访问 MenuCard。指定 MenuCardId 属性来定义一个外键关系(代码文件 MenusSample/Menu.cs)：

```
public class Menu
{
  public int MenuId { get; set; }
  public string Text { get; set; }
  public decimal Price { get; set; }
  public int MenuCardId { get; set; }
  public MenuCard MenuCard { get; set; }
  public override string ToString() => Text;
}
```

到数据库的映射是通过 MenusContext 类实现的。这个类的定义类似于前面的上下文类型；它只包含两个属性，映射两个对象类型：Menus 和 MenuCards 属性(代码文件 MenusSamples/MenusContext.cs)：

```
public class MenusContext: DbContext
{
  private const string ConnectionString = @"server=(localdb)\MSSQLLocalDb;" +
    "Database=MenuCards;Trusted_Connection=True";

  public DbSet<Menu> Menus { get; set; }
  public DbSet<MenuCard> MenuCards { get; set; }

  protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
  {
    base.OnConfiguring(optionsBuilder);
    optionsBuilder.UseSqlServer(ConnectionString);
  }
}
```

在创建代码中修改一些部分是有益的。例如，Text 和 Title 列的尺寸可以减小 NVARCHAR(MAX)中的值。另外，SQL Server 定义了 Money 类型，它可用于 Price 列，模式名称可以在 dbo 中修改。Entity Framework 提供了两个选项——数据注释和流利 API——用于在代码中进行这些修改，如下面所述。

## 26.4.2 数据注释

要影响生成的数据库，一个方法是给实体类型添加数据注释。默认情况下，表的名称来自于上下文的属性。因此要映射 Menu 类，应使用 Menus 表，因为映射 Menu 的 DbSet 属性名为 Menus。有了数据注释，可以使用 Table 特性来改变表格。要改变模式名称，Table 特性定义 Schema 特性。为了给字符串类型指定另一个长度，

可以应用 MaxLength 特性(代码文件 MenusWithDataAnnotations/MenuCard.cs)：

```
[Table("MenuCards", Schema = "mc")]
public class MenuCard
{
  public int MenuCardId { get; set; }

  [MaxLength(120)]
  public string Title { get; set; }
  public List<Menu> Menus { get; } = new List<Menu>();
}
```

> **注意：**
> EF Core 使用上下文中的属性名映射到表格上，这不同于 Entity Framework。Entity Framework 使用一个字典查找单复数名称。

在 Menu 类中，应用了 Table 和 MaxLength 特性。为了更改 SQL 类型，可以使用 Column 特性(代码文件 MenusWithDataAnnotations/Menu.cs)：

```
[Table("Menus", Schema = "mc")]
public class Menu
{
  public int MenuId { get; set; }
  [MaxLength(50)]
  public string Text { get; set; }
  [Column(TypeName ="Money")]
  public decimal Price { get; set; }
  public int MenuCardId { get; set; }
  public MenuCard MenuCard { get; set; }
}
```

应用迁移并创建数据库后，可以在 Title、Text 和 Price 列上看到表的新名称和模式名称，以及改变了的数据类型：

```
CREATE TABLE [mc].[MenuCards] (
  [MenuCardId] INT IDENTITY (1, 1) NOT NULL,
  [Title] NVARCHAR (120) NULL,
  CONSTRAINT [PK_MenuCard] PRIMARY KEY CLUSTERED ([MenuCardId] ASC)
);

CREATE TABLE [mc].[Menus] (
  [MenuId] INT IDENTITY (1, 1) NOT NULL,
  [MenuCardId] INT NOT NULL,
  [Price] MONEY NOT NULL,
  [Text] NVARCHAR (50) NULL,
  CONSTRAINT [PK_Menu] PRIMARY KEY CLUSTERED ([MenuId] ASC),
  CONSTRAINT [FK_Menu_MenuCard_MenuCardId] FOREIGN KEY ([MenuCardId])
  REFERENCES [mc].[MenuCards] ([MenuCardId]) ON DELETE CASCADE
);
```

### 26.4.3　流利 API

影响所创建表的另一种方法是通过 DbContext 派生类的 OnModelCreating 方法使用流利 API。使用它的优点是，实体类型可以很简单，不需要添加任何特性，流利 API 也提供了比应用特性更多的选择。

下面的代码片段显示了 BooksContext 类的 OnModelCreating 方法的重写版本。接收为参数的 ModelBuilder 类提供了一些方法，定义了一些扩展方法。HasDefaultSchema 是一个扩展方法，把默认模式应用于模型，现在用于所有类型。Entity 方法返回一个 EntityTypeBuilder，允许自定义实体，如把它映射到特定的表名，定义键和索引：

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
  base.OnModelCreating(modelBuilder);
  modelBuilder.HasDefaultSchema("mc");
  modelBuilder.Entity<MenuCard>()
    .ToTable("MenuCards")
    .HasKey(c => c.MenuCardId);
  //...
```

```
  modelBuilder.Entity<Menu>()
    .ToTable("Menus")
    .HasKey(m => m.MenuId);
  //...
}
```

EntityTypeBuilder 定义了一个 Property 方法来配置属性。Property 方法返回一个 PropertyBuilder，它允许用最大长度值、需要的设置和 SQL 类型配置属性，指定是否应该自动生成值(例如标识列)：

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
  //...
  modelBuilder.Entity<MenuCard>()
    .Property(c => c.MenuCardId)
    .ValueGeneratedOnAdd();

  modelBuilder.Entity<MenuCard>()
    .Property(c => c.Title)
    .HasMaxLength(50);
  //...

  modelBuilder.Entity<Menu>()
    .Property(m => m.MenuId)
    .ValueGeneratedOnAdd();

  modelBuilder.Entity<Menu>()
    .Property(m => m.Text)
    .HasMaxLength(120);

  modelBuilder.Entity<Menu>()
    .Property(m => m.Price)
    .HasColumnType("Money");
  //...
}
```

要定义一对多映射，EntityTypeBuilder 定义了映射方法。方法 HasMany 与 WithOne 结合，用一个菜单卡定义了到很多菜单的映射。HasMany 需要与 WithOne 链接起来。方法 HasOne 需要和 WithMany 或 WithOne 链接起来。链接 HasOne 与 WithMany，会定义一对多关系；链接 HasOne 与 WithOne，会定义一对一关系：

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
  //...
  modelBuilder.Entity<MenuCard>()
    .HasMany(c => c.Menus)
    .WithOne(m => m.MenuCard);

  modelBuilder.Entity<Menu>()
    .HasOne(m => m.MenuCard)
    .WithMany(c => c.Menus)
    .HasForeignKey(m => m.MenuCardId);
}
```

### 26.4.4 自包含类型的配置

拥有一个更复杂的 DbContext 后，OnModelCreating 方法可能会变得很长。EF Core 2.0 提供了一个新选项，用于为每个类型定义配置类。要创建一个配置类，类需要使用方法 Configure 实现接口 IEntityTypeConfiguration。为 MenuCard 类型创建 MenuCardConfiguration，可以简化配置，如下面的代码片段所示(代码文件 MenusSample/MenuCardConfiguration.cs)：

```
public class MenuCardConfiguration : IEntityTypeConfiguration<MenuCard>
{
  public void Configure(EntityTypeBuilder<MenuCard> builder)
  {
    builder.ToTable("MenuCards")
      .HasKey(c => c.MenuCardId);
    builder.Property(c => c.MenuCardId)
      .ValueGeneratedOnAdd();
    builder.Property(c => c.Title)
      .HasMaxLength(50);

    builder.HasMany(c => c.Menus)
      .WithOne(m => m.MenuCard);
```

```
    }
}
```

类 Menu 的配置在 MenuConfiguration 类中定义。EntityTypeBuilder 使用的方法与前面使用的方法相同。代码更简单，因为实体类型不需要被选中，而有了 IEntityTypeConfiguration，就已经指定了实体类型：

```
public class MenuConfiguration : IEntityTypeConfiguration<Menu>
{
  public void Configure(EntityTypeBuilder<Menu> builder)
  {
    builder.ToTable("Menus")
     .HasKey(m => m.MenuId);

    builder.Property(m => m.MenuId)
      .ValueGeneratedOnAdd();

    builder.Property(m => m.Text)
      .HasMaxLength(120);

    builder.Property(m => m.Price)
      .HasColumnType("Money");

    builder.HasOne(m => m.MenuCard)
      .WithMany(m => m.Menus)
      .HasForeignKey(m => m.MenuCardId);
  }
}
```

MenusContext 类的 OnModelCreating 方法现在可以简化了。要应用 IEntityTypeConfiguration 类型的配置，需要调用 ModelBuilder 的 ApplyConfiguration 方法(代码文件 MenusSample/MenusContext.cs)：

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
  base.OnModelCreating(modelBuilder);
  modelBuilder.HasDefaultSchema("mc");

  modelBuilder.ApplyConfiguration(new MenuCardConfiguration());
  modelBuilder.ApplyConfiguration(new MenuConfiguration());
}
```

### 26.4.5　在数据库中搭建模型

除了从模型中创建数据库之外，也可以从数据库中创建模型。

为此，必须在项目的包列表中添加 NuGet 包 Microsoft.EntityFrameworkCore.Design，在 DotnetCliToolReference 元素中添加 Microsoft.EntityFrameworkCore.Tools.Dotnet。Microsoft.EntityFrameworkCore.Design 包只需要用于项目本身，包需要用于引用这个包的其他项目，所以可以指定 PrivateAssets 特性（项目文件 ScaffoldSample/ScaffoldSample.csproj）：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.EntityFrameworkCore"
      Version="2.0.0" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer"
      Version="2.0.0" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Design"
      Version="2.0.0" PrivateAssets="All" />
  </ItemGroup>
  <ItemGroup>
    <DotNetCliToolReference
      Include="Microsoft.EntityFrameworkCore.Tools.Dotnet"
      Version="2.0.0" />
  </ItemGroup>
</Project>
```

安装了工具后，就可以在 Developer Command Prompt 中启动 dotnet ef 命令：

```
> dotnet ef dbcontext scaffold
"server=(localdb)\MSSQLLocalDb;database=MenuCards;
```

```
trusted_connection=true" "Microsoft.EntityFrameworkCore.SqlServer"
```

dbcontext 命令允许列出项目中的 DbContext 对象，创建 DBContext 对象。scaffold 命令创建 DbContext 派生类以及模型类。dotnet ef dbcontext scaffold 命令需要两个参数：数据库的连接字符串和应该使用的提供程序。前面的语句显示，在 SQL Server (localdb) \ MSSQLLocalDb 上访问数据库 MenuCards。使用的提供程序是 Microsoft.EntityFrameworkCore.SqlServer。这个 NuGet 包需要添加到项目中。

在运行了这个命令之后，可以看到生成的 DbContext 派生类以及模型类型。模型的配置默认使用流利 API 来完成。然而，可以改为使用数据注释，提供--data-annotations 选项。也可以影响生成的上下文类名以及输出目录。使用选项--help 可以查看不同的可用选项。

### 26.4.6 映射到字段

EF Core 不仅允许将表列映射到属性，还允许映射到私有字段。因此可以创建只读属性，使用在类之外无法访问的私有字段。

看看下面代码片段中的类 Book。这个类包含一个私有字段_bookId，该字段只能在类中访问(它是在 ToString 方法中使用的)。Title 是一个读/写属性，Publisher 是一个只读属性。发布者使用字段_publisher。EF Core 在类中需要的是一个默认构造函数，但是这个构造函数可以通过 private 访问修饰符声明(代码文件 BooksSample/Book.cs)：

```
public class Book
{
  // parameterless constructor neeeded for EF Core
  private Book() { }

  public Book(string title, string publisher)
  {
    _title = title;
    _publisher = publisher;
  }

  private int _bookId = 0;
  public string Title { get; set; }
  private string _publisher;
  public string Publisher => _publisher;

  public override string ToString() =>
    $"id: {_bookId}, title: {Title}, publisher: {Publisher}";
}
```

为了避免输入错误，对于列名，定义具有强类型列名的类 ColumnNames。另外，using static 声明访问没有类名的 const 值(代码文件 BooksSample/BooksContext.cs)：

```
using static BooksSample.ColumnNames;

namespace BooksSample
{
  internal class ColumnNames
  {
    public const string LastUpdated = nameof(LastUpdated);
    public const string IsDeleted = nameof(IsDeleted);
    public const string BookId = nameof(BookId);
    public const string AuthorId = nameof(AuthorId);
  }
  //...
}
```

属性 Publisher 现在可以配置为使用 HasField 方法映射到相应的字段。_bookId 没有相应的属性，因此它配置了 Property 方法的一个重载，该方法将名称指定为 string。这将数据库表中的 BookId 列映射到字段_bookId(代码文件 BooksSample/BooksContext.cs)：

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
  base.OnModelCreating(modelBuilder);
  //...

  modelBuilder.Entity<Book>().Property(b => b.Title)
```

```
    .IsRequired()
    .HasMaxLength(50);

  modelBuilder.Entity<Book>().Property(b => b.Publisher)
    .HasField("_publisher")
    .IsRequired(false)
    .HasMaxLength(30);

  modelBuilder.Entity<Book>().Property<int>(BookId)
    .HasField("_bookId")
    .IsRequired();

  modelBuilder.Entity<Book>()
    .HasKey(BookId);
}
```

在创建 Book 对象时，需要使用构造函数。属性没有 set 访问器。初始化 Book 对象后，使用 AddRangeAsync 方法将其添加到 BooksContext 中(代码文件 BooksSample/Program.cs)：

```
private async Task AddBooksAsync()
{
  using (var context = new BooksContext())
  {
    var b1 = new Book("Professional C# 6 and .NET Core 1.0", "Wrox Press");
    var b2 = new Book("Professional C# 5 and .NET 4.5.1", "Wrox Press");
    var b3 = new Book("JavaScript for Kids", "Wrox Press");
    var b4 = new Book("Web Design with HTML and CSS", "For Dummies");
    await context.Books.AddRangeAsync(b1, b2, b3, b4);
    int records = await context.SaveChangesAsync();

    Console.WriteLine($"{records} records added");
  }
  Console.WriteLine();
}
```

## 26.4.7　阴影属性

EF Core 不仅允许将数据库列映射到私有字段，还可以定义一个在模型中根本不显示的映射。可以使用阴影属性，这些属性可以用上下文中的实体来检索，但不能用于模型。

阴影属性定义为字符串。为了避免在多次使用这些字符串时出现拼写错误，指定了一个定义常量字符串的类(代码文件 BooksSample/BooksContext.cs)：

```
public class ColumnNames
{
  public const string LastUpdated = nameof(LastUpdated);
  public const string IsDeleted = nameof(IsDeleted);
  public const string BookId = nameof(BookId);
}
```

要访问类的成员而不使用类名，使用 using static 声明：

```
using static MappingToFields.ColumnNames;
```

下面的代码片段使用前面定义的强类型字符串来定义 IsDeleted 和 LastUpdated 阴影属性：

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
  base.OnModelCreating(modelBuilder);
  //...

  // shadow properties
  modelBuilder.Entity<Book>().Property<bool>(IsDeleted);
  modelBuilder.Entity<Book>().Property<DateTime>(LastUpdated);
}
```

阴影属性 LastUpdated 用于编写实体最后更新的实际时间。IsDeleted 属性用于定义删除实体的状态，而不是删除它。有时，不删除用户请求的数据，而把它标记为已删除是很有用的。这允许执行撤销来恢复实体，并提供历史信息。

要自动更新阴影属性 LastUpdated，需要重写 SaveChangesAsync 方法。如果使用同步 SaveChanges 方法向数据库写入更改，那么也需要重写此方法。在实现代码中，将检查实体的实际状态。如果状态是 Added、Modified

或 Deleted，则使用当前时间更新阴影属性。要管理阴影属性 IsDeleted，删除的实体改为 Modified 状态，IsDeleted 阴影属性设置为 true。阴影属性在允许访问它的模型中没有属性；相反，可以使用 EntityEntry 的 CurrentValues 索引器(代码文件 BooksSample/BooksContext.cs)：

```
public override Task<int> SaveChangesAsync(CancellationToken cancellationToken
  = default)
{
  ChangeTracker.DetectChanges();

  foreach (var item in ChangeTracker.Entries<Book>()
    .Where(e => e.State == EntityState.Added ||
              e.State == EntityState.Modified ||
              e.State == EntityState.Deleted))
  {
    item.CurrentValues[LastUpdated] = DateTime.Now;

    if (item.State == EntityState.Deleted)
    {
      item.State = EntityState.Modified;
      item.CurrentValues[IsDeleted] = true;
    }
  }

  return base.SaveChangesAsync(cancellationToken);
}
```

**注意：**
示例代码使用的更改跟踪器参见“对象跟踪”一节。

**注意：**
有了 IsDeleted 属性，在使用正常查询时，最好不返回设置了 IsDeleted 属性的实体。而可以使用 EF Core 2.0 特性——全局查询过滤器来实现这一点，该特性将在后面的小节中讨论。

为了显示已删除的实体，定义了 DeleteBookAsync 方法，该方法使用传递给该方法的 ID 来删除实体。在这里，通过传递实体对象来调用 Remove 方法，并调用 SaveChanges(代码文件 BooksSample/Program.cs)：

```
private async Task DeleteBookAsync(int id)
{
  using (var context = new BooksContext())
  {
    Book b = await context.Books.FindAsync(id);
    if (b == null) return;

    context.Books.Remove(b);
    int records = await context.SaveChangesAsync();
    Console.WriteLine($"{records} books deleted");
  }
  Console.WriteLine();
}
```

在幕后，由于对 SaveChangesAsync 方法的更改而设置了 IsDeleted 阴影属性。要验证这一点，可以使用方法 EF.Property，通过传递 IsDeleted 字符串，来访问阴影属性。所有带有此标志的 Book 实体都显示在 QueryDeletedBooksAsync 方法中：

```
private async Task QueryDeletedBooksAsync()
{
  using (var context = new BooksContext())
  {
    IEnumerable<Book> deletedBooks =
      await context.Books
        .Where(b => EF.Property<bool>(b, IsDeleted))
        .ToListAsync();

    foreach (var book in deletedBooks)
    {
      Console.WriteLine($"deleted: {book}");
    }
  }
}
```

**注意：**

EF 是 Microsoft.EntityFrameworkCore 名称空间中的一个静态类，在 EF 类型不可用时，它提供了有用的静态方法。本节介绍了可以用于访问阴影状态的 Property 方法。在本章后面，EF 类与编译的查询和 EF.Functions 一起使用。

# 26.5　查询

前面定义了模型，下面了解查询的更多细节。本节讨论：

- 基本查询
- 在服务器和客户端上评价
- 原始 SQL 查询
- 编译过的查询有更好的性能
- 全局查询过滤器
- EF.Functions

## 26.5.1　基本查询

如前所述，访问 DbSet 的上下文属性将返回指定表的所有实体的列表。下面详细讨论。

访问 Books 属性，会从数据库中检索所有 Book 记录(代码文件 BooksSample/QuerySamples.cs)：

```
private async Task QueryAllBooksAsync()
{
  Console.WriteLine(nameof(QueryAllBooksAsync));
  using (var context = new BooksContext())
  {
    List<Book> books = await context.Books.ToListAsync();
    foreach (var b in books)
    {
      Console.WriteLine(b);
    }
  }
  Console.WriteLine();
}
```

有了异步 API，也可以使用从 ToAsyncEnumerable 方法返回的 IAsyncEnumerable 接口，使用 ForEachAsync 方法而不是 foreach 循环：

```
await context.Books.ToAsyncEnumerable()
  .ForEachAsync(b =>
  {
    Console.WriteLine(b);
  });
```

访问 Books 属性，会把下面的 SQL 语句发送到数据库：

```
SELECT  [b].[BookId], [b].[IsDeleted], [b].[LastUpdated], [b].[Publisher],
        [b].[Title]
FROM [Books] AS [b]
```

可以使用 Find 和 FindAsync 方法查询具有特定键的对象。如果没有找到记录，该方法就返回 null：

```
Book b = await context.Books.FindAsync(id);
if (b != null)
{
  //...
```

这就得到了一个带有 TOP(1)和 WHERE 子句的 SELECT SQL 语句：

```
SELECT TOP(1)  [e].[BookId], [e].[IsDeleted], [e].[LastUpdated],
               [e].[Publisher], [e].[Title]
FROM [Books] AS [e]
WHERE [e].[BookId] = @__get_Item_0
```

与使用 Find 方法不同，还可以使用同步的 Single 或 SingleOrDefault 方法，或者使用异步变体 SingleAsync

或 SingleOrDefaultAsync。Single 和 SingleOrDefault 的区别在于，Single 在没有找到记录时抛出异常，而 SingleOrDefault 会在没有找到记录时返回 null。这些方法还在找到多个记录时抛出一个异常。

下面的代码片段使用 SingleOrDefaultAsync 方法来请求书名：

```
Book book = await context.Books.SingleOrDefaultAsync(b => b.Title == title);
```

生成的 SQL 语句要求 TOP(2)记录，它允许在找到两个记录时抛出异常：

```
SELECT TOP(2)  [b].[BookId], [b].[IsDeleted], [b].[LastUpdated],
               [b].[Publisher], [b].[Title]
FROM [Books] AS [b]
WHERE [b].[Title] = @__title_0
```

Where 方法允许基于条件进行简单的过滤。还可以在 Where 表达式中使用 Contains 方法。Where 方法没有可用的异步变体，因为 Where 方法使用了惰性求值。可以使用 foreach 语句来迭代查询的所有结果。然而，foreach 会触发查询的执行，阻塞线程，直到检索到结果。与使用 foreach 和 Where 方法的结果不同，可以使用 ToListAsync 立即触发执行，但要在任务中执行：

```
List<Book> wroxBooks = await context.Books
  .Where(b => b.Title.Contains(title))
  .ToListAsync();
```

生成的 SQL 语句使用了 SQL 子句中一个简单的 WHERE：

```
SELECT  [b].[BookId], [b].[IsDeleted], [b].[LastUpdated], [b].[Publisher],
        [b].[Title]
FROM [Books] AS [b]
WHERE (CHARINDEX(@__title_0, [b].[Title]) > 0) OR (@__title_0 = N'')
```

在第 12 章中详细介绍了更多的 LINQ 方法和 LINQ 子句，也可以在 EF Core 中使用它们。请记住，LINQ to Objects 和 LINQ to EF Core 的实现是不同的。在 LINQ to EF Core 中，使用表达式树可以在运行时使用完整的 LINQ 表达式创建 SQL 查询。在 LINQ to Objects 中，大多数 LINQ 查询都是在 Enumerable 类中定义的。带有表达式树的 LINQ 在 Queryable 类中实现，对 EF Core(如异步变体)的许多增强在 EntityFrameworkQueryableExtensions 类中实现。有关表达式树的更多信息，请参见第 12 章。

### 26.5.2 客户端和服务器求值

不是查询的每个部分都可以转换为 SQL 语句，从而在服务器上运行。有些部分需要在客户端上运行。EF Core 允许进行透明的客户端和服务器求值。如果查询不能解析，会自动在客户端上运行。这对于使用不同的提供程序有很大的优势。例如，对于一个提供程序，可以在服务器上对查询进行完全的求值。使用不转换所有查询的另一个提供程序，程序仍然运行，但是有些部分现在在客户端上进行求值。

下面看一个 n-n 关系的示例。Book 类型与 Author 类型通过一个关联实体关联。一本书可以由多名作者写，一个作者也可以写多本书。

下面的代码片段通过 Books 属性访问 Book 对象。Where 方法用于过滤，OrderBy 方法定义顺序。使用 Select 方法定义结果——包括使用 BookAuthors 属性与作者关联：

```
var books = context.Books
  .Where(b => b.Title.StartsWith("Pro"))
  .OrderBy(b => b.Title)
  .Select(b => new
  {
    b.Title,
    Authors = b.BookAuthors
  });
```

所有这些都使用 EF Core 2.0 转换为 SQL 语句。求值完全在服务器上进行，使用 Select、INNER JOIN、Where 和 ORDER BY，通过关联转换 Where、OrderBy 和 Select：

```
SELECT [b.BookAuthors].[BookId], [b.BookAuthors].[AuthorId]
FROM [BookAuthors] AS [b.BookAuthors]
INNER JOIN (
  SELECT [b0].[BookId], [b0].[Title]
  FROM [Books] AS [b0]
  WHERE [b0].[Title] LIKE N'Pro' + N'%' AND (LEFT([b0].[Title], LEN(N'Pro')) =
```

```
    N'Pro')
) AS [t] ON [b.BookAuthors].[BookId] = [t].[BookId]
ORDER BY [t].[Title], [t].[BookId]
```

如果将 Select 语句修改为返回包含作者的逗号分隔字符串，那么结果将非常不同。这在下面的代码片段中完成：把一个字符串分配给 Authors 属性。使用关系 BookAuthors，只选择作者的 FirstName 和 LastName 属性，string.Join 把列表连接到一个字符串上(代码文件 BooksSample/QuerySamples.cs)：

```
var books = context.Books
  .Where(b => b.Title.StartsWith("Pro"))
  .OrderBy(b => b.Title)
  .Select(b => new
  {
    b.Title,
    Authors = string.Join(", ", b.BookAuthors.Select(a =>
      $"{a.Author.FirstName} {a.Author.LastName}").ToArray())
  });
```

EF Core 2.0 无法将此查询转换为 SQL 语句。来自 EF Core 的日志信息显示了这个警告：

```
warn: Microsoft.EntityFrameworkCore.Query[200500]
  The LINQ expression 'join Author a.Author in value(
    Microsoft.EntityFrameworkCore.Query.Internal.EntityQueryable`1[
      BooksSample.Author]) on Property([a], "AuthorId") equals
      Property([a.Author], "AuthorId")'
  could not be translated and will be evaluated locally.
```

现在执行三个查询。这些查询的结果在客户端上连接。应用程序仍然可以工作，但是查询并没有那么有效。三个语句在 SQL Server 中执行，而不是执行一个语句。分析查询时，可以看到在客户机上执行求值之前，所有的作者都是从服务器中检索的。这可能会导致向客户端的大量转移：

```
SELECT [b].[Title], [b].[BookId]
FROM [Books] AS [b]
WHERE [b].[Title] LIKE N'Pro' + N'%' AND (LEFT([b].[Title], LEN(N'Pro')) =
  N'Pro')
ORDER BY [b].[Title]

SELECT [b0].[BookId], [b0].[AuthorId]
FROM [BookAuthors] AS [b0]
WHERE @_outer_BookId = [b0].[BookId]

SELECT [a.Author].[AuthorId], [a.Author].[FirstName], [a.Author].[LastName]
FROM [Authors] AS [a.Author]
```

自动进行客户端和服务器的求值是很实用的。与 EF Core 1.0 不同，用于 EF Core 2.0 的 SQL Server 提供程序可以在服务器上进行更多的求值，未来的版本甚至可能在服务器上支持更多的求值。使用其他提供程序可能会有不同的结果。效率是不同的，但至少程序是有效的。

为了避免在服务器上进行求值，可以配置上下文，使求值仅在服务器上进行时抛出异常。为此，在配置上下文时，可以在 optionsBuilder 上调用 ConfigureWarnings 方法：

```
optionsBuilder.UseSqlServer(ConnectionString)
  .ConfigureWarnings(warnings =>
    warnings.Throw(RelationalEventId.QueryClientEvaluationWarning));
```

**警告：**

客户端和服务器求值是一个很好的特性，可以使程序在不同的提供程序之间工作。然而，这会导致性能损失。要为了获得最佳性能而定义查询，可以通过配置抛出异常，来发现在客户端进行求值的情况。然后可以相应地更改查询。

## 26.5.3　原始 SQL 查询

EF Core 2.0 还允许定义原始 SQL 查询，原始 SQL 查询返回实体对象并跟踪这些对象。只需要调用 DbSet 对象的 FromSql 方法，如下面的代码片段所示(代码文件 BooksSample/QuerySamples.cs)：

```
private async Task RawSqlQuery(string publisher)
{
  Console.WriteLine(nameof(RawSqlQuery));
```

```
    using (var context = new BooksContext())
    {
      IList<Book> books = await context.Books.FromSql(
        $"SELECT * FROM Books WHERE Publisher = {publisher}")
        .ToListAsync();

      foreach (var b in books)
      {
        Console.WriteLine($"{b.Title} {b.Publisher}");
      }
    }
    Console.WriteLine();
  }
```

分配给 RawSql 方法的 SQL 查询需要返回作为模型一部分的实体类型，需要返回模型所有属性的数据。

分配给 FromSql 方法的 SQL 字符串可能看起来像 SQL 注入，因为字符串被定义了。然而，事实并非如此。FromSql 要求分配一个 FormattableString 类型。对于这个 FormattableString，EF Core 提取参数并创建 SQL 参数。

**注意：**
有关字符串插值和 FormattableString 类型的更多信息，请参阅第 9 章。

### 26.5.4 已编译查询

对于需要反复执行的查询，可以创建一个只需要执行的已编译查询。可以使用 EF.CompileQuery 创建已编译的查询。此方法提供了不同的泛型重载，可以在其中传递不同数量的参数。在下面的代码片段中，创建一个查询来定义一个字符串参数。该方法需要一个委托参数，其中第一个参数是类型 BooksContext，第二个参数是字符串——这里使用的是 publisher。定义已编译的查询后，可以使用它传递上下文和参数(代码文件 BooksSample/QuerySamples.cs)：

```
private void CompiledQuery()
{
  Console.WriteLine(nameof(CompiledQuery));
  Func<BooksContext, string, IEnumerable<Book>> query =
    EF.CompileQuery<BooksContext, string, Book>((context, publisher) =>
      context.Books.Where(b => b.Publisher == publisher));

    using (var context = new BooksContext())
    {
      IEnumerable<Book> books = query(context, "Wrox Press");

      foreach (var b in books)
      {
        Console.WriteLine($"{b.Title} {b.Publisher}");
      }
    }
    Console.WriteLine();
  }
```

可以为成员字段创建一个已编译的查询，以便在需要的时候使用它，并且可以根据需要，传递不同的上下文，调用查询。

### 26.5.5 全局查询过滤器

本章的前面介绍了使用 IsDeleted 列的阴影状态。不需要为每个查询定义 WHERE 子句，以避免返回 IsDeleted 为真的记录；相反，可以在创建模式时定义全局查询过滤器。这是下一个代码片段所做的——全局检查 IsDeleted。因为 IsDeleted 并没有映射到模型，而只是通过阴影状态来检查，所以可以使用 EF.Property 检索值(代码文件 BooksSample/BooksContext.cs)：

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
  base.OnModelCreating(modelBuilder);

  modelBuilder.Entity<Book>().HasQueryFilter(
    b => !EF.Property<bool>(b, IsDeleted));
  //...
}
```

在定义了这个查询过滤器之后，在该上下文使用的每个查询中都添加了对 IsDeleted 的 WHERE 检查。

**注意：**

全局查询过滤器也适用于多租户需求。可以为特定的 tenant-id 筛选上下文的所有查询。在构建上下文时，只需要传递 tenant-id。不使用依赖注入，可以将 tenant-id 传递给构造函数。使用依赖注入，只需要指定一个用构造函数注入的服务，其中，可以在查询过滤器中检索 tenant-id。

**注意：**

可以忽略全局查询过滤器。例如，要获取所有被删除的实体，可以使用带有 LINQ 表达式的 IgnoreQueryFilters 方法。

### 26.5.6　EF.Functions

EF Core 允许自定义扩展方法可以由提供程序实现。为此，EF 类定义了 DbFunctions 类型的 Functions 属性，它可以使用扩展方法进行扩展。在撰写本文时，Like 方法是关系数据提供程序的这样一种扩展。

下面的代码片段使用 EF.Functions.Like，并提供包含参数 titleSegment 的表达式，增强了 Where 方法的查询。参数 titleSegment 嵌入在两个%字符内(代码文件 BooksSample/QuerySample.cs)：

```
public static async Task UseEFCunctions(string titleSegment)
{
  Console.WriteLine(nameof(UseEFCunctions));
  using (var context = new BooksContext())
  {
    string likeExpression = $"%{titleSegment}%";

    IList<Book> books = await context.Books.Where(
      b => EF.Functions.Like(b.Title, likeExpression)).ToListAsync();
    foreach (var b in books)
    {
      Console.WriteLine($"{b.Title} {b.Publisher}");
    }
  }
  Console.WriteLine();
}
```

运行应用程序时，包含 EF.Functions.Like 的 Where 方法转换为带有 LIKE 的 SQL 子句 WHERE：

```
SELECT [b].[BookId], [b].[IsDeleted], [b].[LastUpdated], [b].[Publisher],
  [b].[Title]
FROM [Books] AS [b]
WHERE ([b].[IsDeleted] = 0) AND [b].[Title] LIKE @__likeExpression_1
```

## 26.6　关系

关系可以定义为一对一或一对多。对于多对多关系，在 EF Core 2.0 中，需要在关系中指定一个中间类，从而将该关系分割为一对多关系和多对一关系。

关系可以使用约定、注释和流利 API 来指定。下一节将讨论这三种变体。

### 26.6.1　使用约定的关系

定义关系的第一个方法是使用约定。下面看一下使用 Book 和 Chapter 类型的例子。一本书可以有多个章节；因此，这是一对多关系。Book 类型还定义了与作者的关系。这里，作者由 User 类表示。稍后，使用注释定义关系时，会解释这个名称的原因。书与作者定义了一对一的关系。(对于有多个作者的书，书中指定的作者是主要作者。)

书是由 Book 类定义的。这个类有一个主键 BookId，它是根据其名称而创建的。关系由 Chapters 属性定义。Chapters 属性为 List<Chapter>类型；这就是 Book 类型定义一对多关系所需的全部内容。书与作者的关系由类型 User 的 Author 属性指定。还可以定义该类型的 AuthorId 属性来指定外键，该属性与 User 类中的键相同。如果没有这个定义，就会创建一个阴影属性(代码文件 RelationUsingConventions/Book.cs)：

```
public class Book
{
  public int BookId { get; set; }
  public string Title { get; set; }
  public List<Chapter> Chapters { get; } = new List<Chapter>();
  public User Author { get; set; }
}
```

**注意：**

*Chapter 属性的另一种可能实现是定义 List<Chapter>类型的读/写属性，而不需要事先创建实例。有了这样的实现，实例将自动从 EF Core 上下文中创建。*

章节由 Chapter 类定义。使用 Book 属性定义关系。一对多关系定义了一方的集合(Book 定义了 Chapter 对象的集合)，和另一方的简单关系(Chapter 定义了一个简单的属性 Book)。使用一章的 Book 属性，可以直接访问相关图书。对于这种类型，BookId 属性指定 Book 的外键。正如 Book 类型所述，如果未将 BookId 指定为类的成员，则通过约定创建阴影属性(代码文件 RelationUsingConventions/Chapter.cs)：

```
public class Chapter
{
  public int ChapterId { get; set; }
  public int Number { get; set; }
  public string Title { get; set; }
  public int BookId { get; set; }
  public Book Book { get; set; }
}
```

User 类定义了主键的 UserId 属性、关系的 Name 属性和 AuthoredBooks 属性 (代码文件 RelationUsingConventions/User.cs)：

```
public class User
{
  public int UserId { get; set; }
  public string Name { get; set; }
  public List<Book> AuthoredBooks { get; set; }
}
```

上下文只需要使用 DbSet<T>类型的属性为 Book、Chapter 和 User 类型指定属性(代码文件 RelationUsingConventions/BooksContext.cs)：

```
public class BooksContext : DbContext
{
  private const string ConnectionString =
    @"server=(localdb)\MSSQLLocalDb;database=Books;trusted_connection=true";

  protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
  {
    base.OnConfiguring(optionsBuilder);

    optionsBuilder.UseSqlServer(ConnectionString);
  }
  public DbSet<Book> Books { get; set; }
  public DbSet<Chapter> Chapters { get; set; }
  public DbSet<User> Users { get; set; }
}
```

**注意：**

*可下载的示例代码包含生成数据库的代码和生成示例数据的代码。因为这段代码与本章前面的代码非常相似，所以后面的示例并没有特别涉及它。更多信息请参考可下载的代码示例。*

在启动应用程序时，使用按照约定定义的映射创建数据库。图 26-2 显示了 Books、Chapters 和 Users 表及其关系。Book 类不为 Users 类型定义外键属性，而是创建一个 AuthorUserId 填充阴影属性：

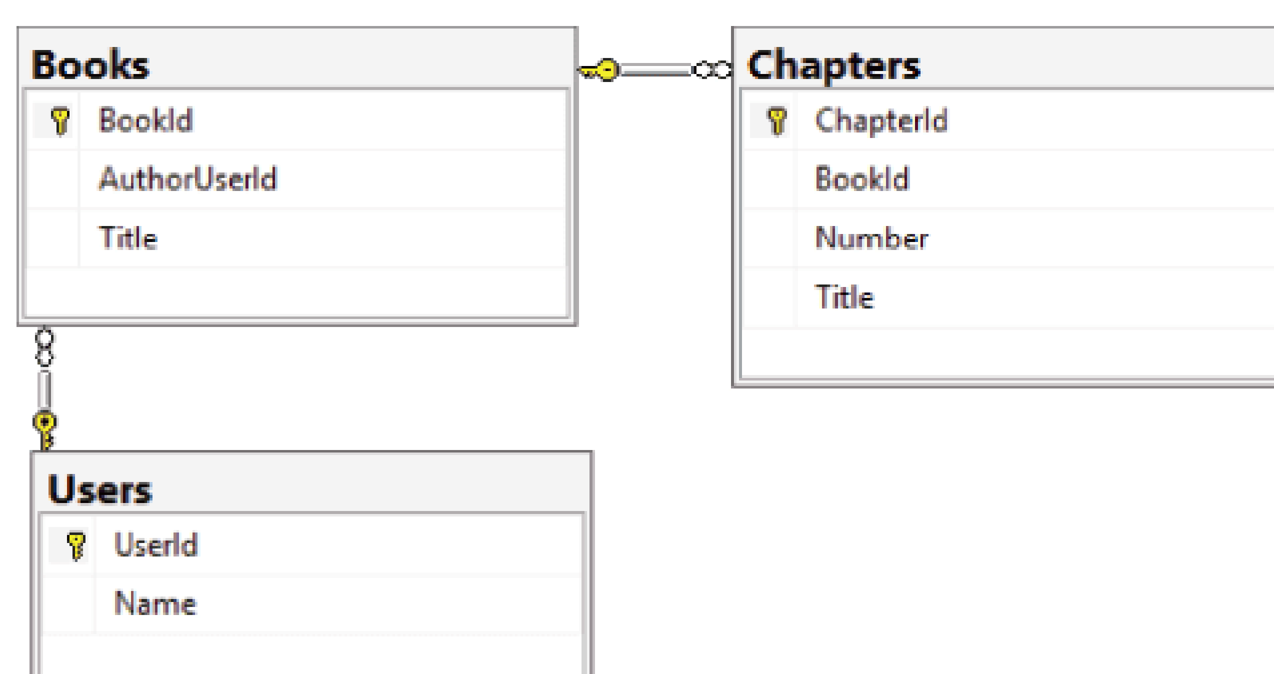



图 26-2

在介绍定义关系的不同方法(使用注释和流利 API)之前，先看看如何使用查询访问相关数据。

## 26.6.2　显式加载相关数据

如果查询书籍并希望显示相关属性(例如，相关章节和相关作者)，则可以使用关系的显式加载。

请看下面的代码片段。查询请求所有具有指定标题的图书，并且只需要一个记录。如果试图在启动查询之后访问所得图书的 Chapters 和 Author 属性，那么这些属性的值为 null。关系不是隐式加载的。EF Core 使用上下文的 Entry 方法来支持显式加载，该方法通过传递一个实体来返回 EntityEntry 对象。EntityEntry 类定义了允许显式加载关系的 Collection 和 Reference 方法。对于一对多关系，可以使用 Collection 方法来指定集合，而一对一关系需要 Reference 方法来指定关系。使用 Load 方法进行显式加载(代码文件 RelationUsingConventions/Program.cs)：

```
private static void ExplicitLoading()
{
  Console.WriteLine(nameof(ExplicitLoading));
  using (var context = new BooksContext())
  {
    var book = context.Books
      .Where(b => b.Title.StartsWith("Professional C# 7"))
      .FirstOrDefault();
    if (book != null)
    {
      Console.WriteLine(book.Title);
      context.Entry(book).Collection(b => b.Chapters).Load();
      context.Entry(book).Reference(b => b.Author).Load();
      Console.WriteLine(book.Author.Name);
      foreach (var chapter in book.Chapters)
      {
        Console.WriteLine($"{chapter.Number}. {chapter.Title}");
      }
    }
  }
  Console.WriteLine();
}
```

实现 Load 方法的 NavigationEntry 类也实现了 IsLoaded 属性，可以在其中检查关系是否已经加载。在调用 Load 方法之前，不需要检查加载的关系；在调用 Load 方法时，如果关系已经加载，就不会再次查询数据库。

当对图书的查询运行应用程序时，下面的 SELECT 语句将在 SQL Server 上执行。此查询仅访问 Books 表：

```
SELECT TOP(1) [b].[BookId], [b].[AuthorUserId], [b].[Title]
FROM [Books] AS [b]
WHERE [b].[Title] LIKE N'Professional C# 7' + N'%' AND (LEFT([b].[Title],
LEN(N'Professional C# 7')) = N'Professional C# 7')
```

使用以下 Load 方法检索书中的章节时，SELECT 语句基于图书 ID 检索章节：

```
SELECT [e].[ChapterId], [e].[BookId], [e].[Number], [e].[Title]
FROM [Chapters] AS [e]
WHERE [e].[BookId] = @__get_Item_0
```

使用第三个查询，从 Users 表中检索用户信息：

```
SELECT [e].[UserId], [e].[Name]
FROM [Users] AS [e]
```

```
WHERE [e].[UserId] = @__get_Item_0
```

EF Core 除了显式地加载相关数据，导致向 SQL Server 发送多个查询之外，还支持即时加载，如下所示。

### 26.6.3 即时加载相关数据

当执行查询时，可以通过调用 Include 方法并指定关系，来立即加载相关数据。下面的代码片段包括成功应用 Where 表达式的书中的章节和作者(代码文件 RelationUsingConventions/Program.cs)：

```
private static void EagerLoading()
{
  Console.WriteLine(nameof(EagerLoading));
  using (var context = new BooksContext())
  {
    var book = context.Books
      .Include(b => b.Chapters)
      .Include(b => b.Author)
      .Where(b => b.Title.StartsWith("Professional C# 7"))
      .FirstOrDefault();
    if (book != null)
    {
      Console.WriteLine(book.Title);

      foreach (var chapter in book.Chapters)
      {
        Console.WriteLine($"{chapter.Number}. {chapter.Title}");
      }
    }
  }
  Console.WriteLine();
}
```

使用 Include，只需要执行一条 SQL 语句来访问 Books 表，并连接 Chapters 和 Users 表：

```
SELECT [b.Chapters].[ChapterId], [b.Chapters].[BookId], [b.Chapters].[Number],
  [b.Chapters].[Title]
FROM [Chapters] AS [b.Chapters]
INNER JOIN (
  SELECT DISTINCT [t].*
  FROM (
    SELECT TOP(1) [b0].[BookId]
    FROM [Books] AS [b0]
    LEFT JOIN [Users] AS [b.Author0] ON [b0].[AuthorUserId] =
      [b.Author0].[UserId]
    WHERE [b0].[Title] LIKE N'Professional C# 7' + N'%' AND (LEFT([b0].[Title],
      LEN(N'Professional C# 7')) = N'Professional C# 7')
    ORDER BY [b0].[BookId]
  ) AS [t]
) AS [t0] ON [b.Chapters].[BookId] = [t0].[BookId]
ORDER BY [t0].[BookId]
```

如果需要包含多个层次的关系，那么方法 ThenInclude 可以用于 Include 方法的结果。

### 26.6.4 使用注释的关系

与使用约定不同，实体类型可以通过应用关系信息进行注释。下面向关系属性添加 ForeignKey 属性，来修改先前创建的 Book 类型，并指定表示外键的属性。在这里，Book 不仅与该书的作者有关联，也与审稿人和项目编辑有关联。这些关系映射到 User 类型。外键属性定义为 int 类型吗？让它们变成可选项。使用强制关系，EF Core 创建级联删除；删除 Book 时，相关的作者、编辑和审稿人也被删除(代码文件 RelationUsingAnnotations/Book.cs)：

```
public class Book
{
  public int BookId { get; set; }
  public string Title { get; set; }
  public List<Chapter> Chapters { get; } = new List<Chapter>();

  public int? AuthorId { get; set; }
  [ForeignKey(nameof(AuthorId))]
  public User Author { get; set; }
  public int? ReviewerId { get; set; }
  [ForeignKey(nameof(ReviewerId))]
  public User Reviewer { get; set; }
```

```
  public int? ProjectEditorId { get; set; }
  [ForeignKey(nameof(ProjectEditorId))]
  public User ProjectEditor { get; set; }
}
```

User 类现在与 Book 类型有多个关联。WrittenBooks 属性列出了添加为 author 的用户的所有图书。类似地，ReviewedBooks 和 EditedBooks 属性与 Book 类型仅在 Reviewer 和 ProjectEditor 属性上相关联。如果相同类型之间存在多个关系，则需要使用 InverseProperty 特性对属性进行注释。使用这个特性，指定关系另一端的相关属性(代码文件 RelationUsingAnnotations/User.cs)：

```
public class User
{
  public int UserId { get; set; }
  public string Name { get; set; }
  [InverseProperty("Author")]
  public List<Book> WrittenBooks { get; set; }
  [InverseProperty("Reviewer")]
  public List<Book> ReviewedBooks { get; set; }
  [InverseProperty("ProjectEditor")]
  public List<Book> EditedBooks { get; set; }
}
```

图 26-3 显示了在 SQL Server 中生成的表的关系。Books 表与 Chapters 表关联，如前面的示例所示。现在，Users 表与 Books 表有三个关联。

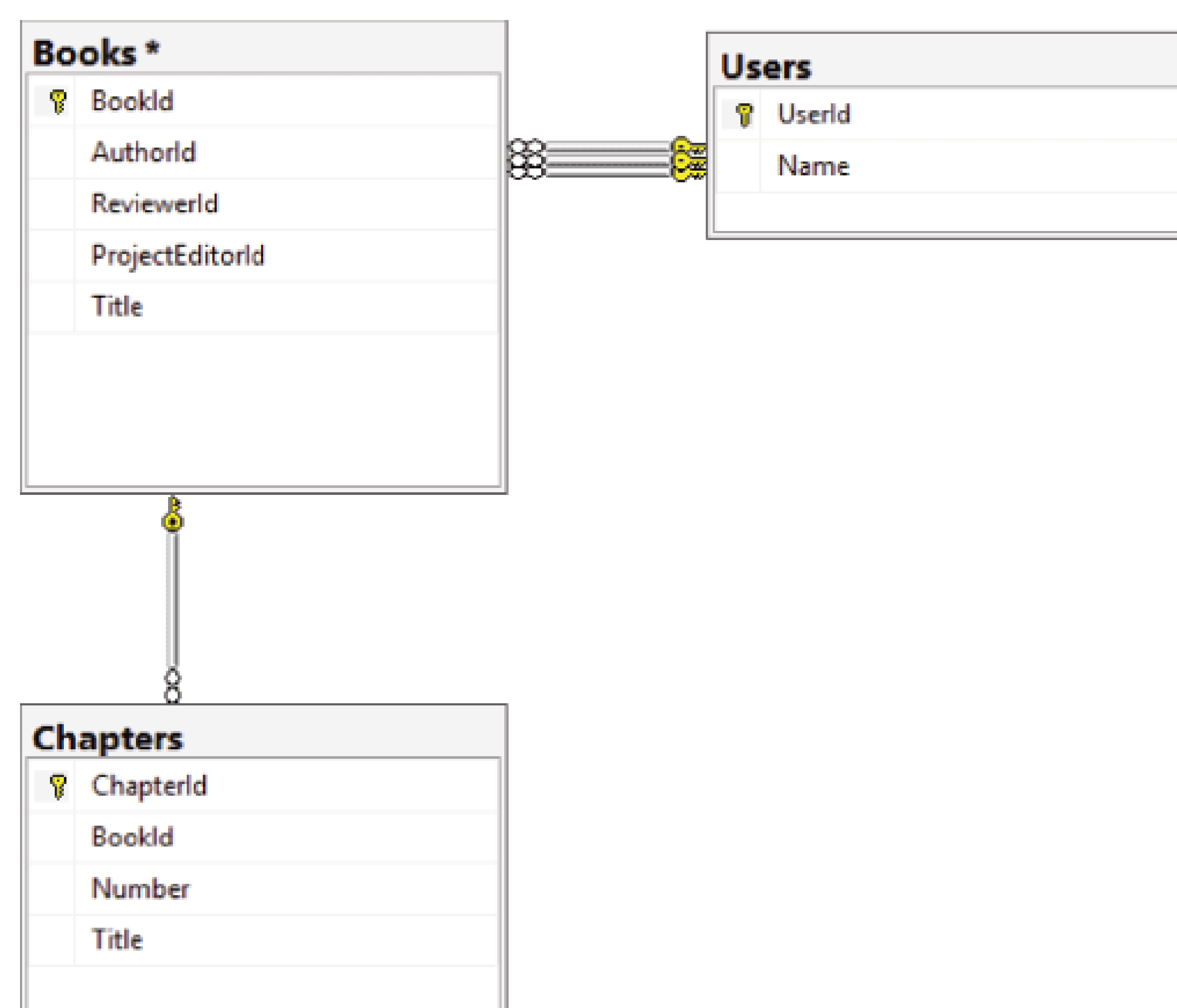



图 26-3

## 26.6.5　使用流利 API 的关系

指定关系的最强大方法是使用流利 API。在流利 API 中，使用 HasOne 和 WithOne 方法定义一对一关系，用 HasOne 和 WithMany 方法定义一对多关系，而多对一关系由 HasMany 和 WithOne 定义。

对于下面的代码示例，模型类型不包括数据库模式上的任何注释。Book 类是一个简单的 POCO 类型，它定义了图书信息的属性，包括关系属性(代码文件 RelationUsingFluentAPI/Book.cs):：

```
public class Book
{
  public int BookId { get; set; }
  public string Title { get; set; }
  public List<Chapter> Chapters { get; } = new List<Chapter>();

  public User Author { get; set; }
  public User Reviewer { get; set; }
  public User Editor { get; set; }
}
```

User 类型的定义也是类似的。除了具有 Name 属性外，User 类型还定义了与 Book 类型的三种不同关系(代码文件 RelationUsingFluentAPI/User.cs)：

```
public class User
{
  public int UserId { get; set; }
  public string Name { get; set; }
  public List<Book> WrittenBooks { get; set; }
  public List<Book> ReviewedBooks { get; set; }
  public List<Book> EditedBooks { get; set; }
}
```

Chapter 类与 Book 类有关系。然而，Chapter 类与 Book 类不同，因为 Chapter 类还定义了一个属性来关联一个外键：BookId(代码文件 RelationUsingFluentAPI/Chapter.cs)：

```
public class Chapter
{
  public int ChapterId { get; set; }
  public int Number { get; set; }
  public string Title { get; set; }
  public int BookId { get; set; }
  public Book Book { get; set; }
}
```

模型类型之间的映射现在在 BooksContext 的 OnModelCreating 方法中定义。Book 类与多个 Chapter 对象相关联;这是使用 HasMany 和 WithOne 定义的。Chapter 类与一个 Book 对象相关联;这是使用 HasOne 和 WithMany 定义的。因为在 Chapter 类中还有一个外键属性，所以使用 HasForeignKey 方法来指定这个键(代码文件 RelationUsingFluentAPI/BooksContext.cs)：

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
  modelBuilder.Entity<Book>()
    .HasMany(b => b.Chapters)
    .WithOne(c => c.Book);
  modelBuilder.Entity<Book>()
    .HasOne(b => b.Author)
    .WithMany(a => a.WrittenBooks);
  modelBuilder.Entity<Book>()
    .HasOne(b => b.Reviewer)
    .WithMany(r => r.ReviewedBooks);
  modelBuilder.Entity<Book>()
    .HasOne(b => b.Editor)
    .WithMany(e => e.EditedBooks);

  modelBuilder.Entity<Chapter>()
    .HasOne(c => c.Book)
    .WithMany(b => b.Chapters)
    .HasForeignKey(c => c.BookId);

  modelBuilder.Entity<User>()
    .HasMany(a => a.WrittenBooks)
    .WithOne(b => b.Author);
  modelBuilder.Entity<User>()
    .HasMany(r => r.ReviewedBooks)
    .WithOne(b => b.Reviewer);
  modelBuilder.Entity<User>()
    .HasMany(e => e.EditedBooks)
    .WithOne(b => b.Editor);
}
```

### 26.6.6 根据约定的每个层次结构的表

EF Core 还支持每个层次结构中表(Table Per Hierarchy，TPH)的关系类型。使用这种关系，形成层次结构的多个模型类用于映射到单个表。这种关系可以使用约定和流利 API 来指定。

下面开始使用约定和形成层次结构的类型 Payment、CashPayment 和 CreditcardPayment，如图 26-4 所示。Payment 是一个基类；CashPayment 和 CreditcardPayment 均派生于它。

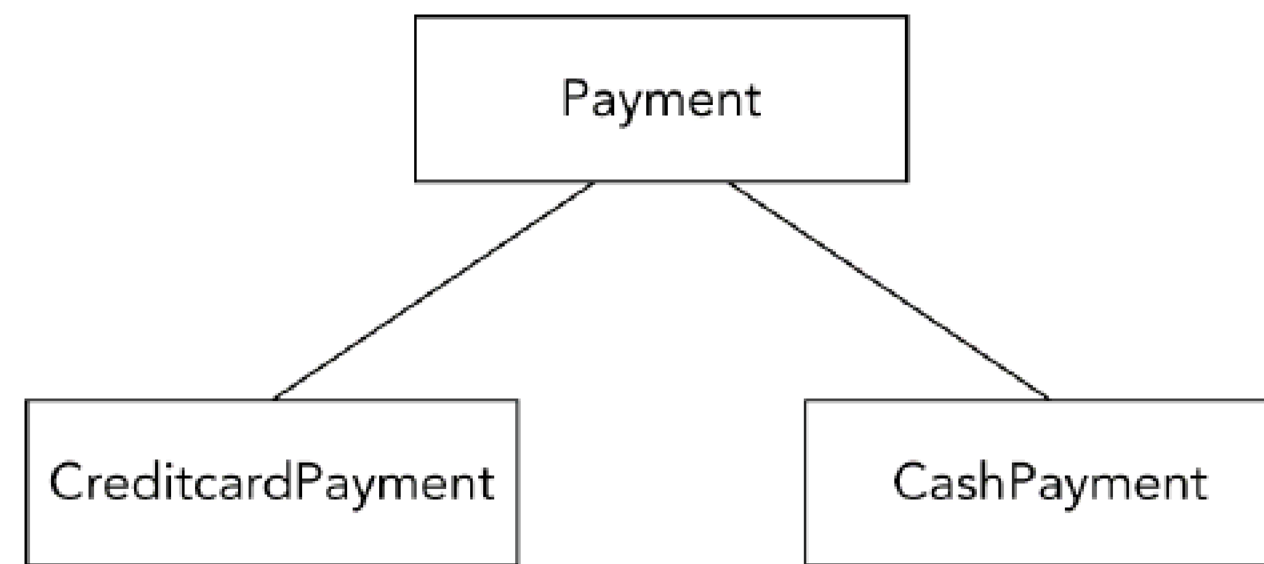



图 26-4

在实现代码中，Payment 类定义了主键，其中包括 PaymentId 属性、所需的 Name 和 Amount 属性。Amount 属性映射到数据库中的一个列类型 Money(代码文件 TPHWithConventions/Payment.cs)：

```
public class Payment
{
  public int PaymentId { get; set; }

  [Required]
  public string Name { get; set; }
  [Column(TypeName = "Money")]
  public decimal Amount { get; set; }
}
```

CreditcardPayment 类派生自 Payment，还添加了 CreditcardNumber 属性(代码文件 TPHWithConventions/CreditcardPayment.cs)：

```
public class CreditcardPayment : Payment
{
  public string CreditcardNumber { get; set; }
}
```

最后，CashPayment 类派生自 Payment，但不声明任何其他成员(代码文件 TPHWithConventions/CashPayment.cs)：

```
public class CashPayment : Payment
{
}
```

EF Core 的上下文类 BankContext 为层次结构中的每个类定义了 DbSet 属性(代码文件 TPHWithConventions/BankContext.cs)：

```
public class BankContext : DbContext
{
  private const string ConnectionString = @"server=(localdb)\MSSQLLocalDb;" +
    "Database=LocalBank;Trusted_Connection=True";

  protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
  {
    base.OnConfiguring(optionsBuilder);

    optionsBuilder.UseSqlServer(ConnectionString);
  }

  public DbSet<Payment> Payments { get; set; }
  public DbSet<CreditcardPayment> CreditcardPayments { get; set; }
  public DbSet<CashPayment> CashPayments { get; set; }
}
```

创建的示例数据定义了两个 CashPayment 和一个 CreditcardPayment 支付(代码文件 TPHWithConventions/Program.cs)：

```
private static void AddSampleData()
{
  using (var context = new BankContext())
  {
    context.CashPayments.Add(
      new CashPayment { Name = "Donald", Amount = 0.5M });
    context.CashPayments.Add(
      new CashPayment { Name = "Scrooge", Amount = 20000M });
    context.CreditcardPayments.Add(
      new CreditcardPayment
      {
```

```
        Name = "Gus Goose",
        Amount = 300M,
        CreditcardNumber = "987654321"
      });
    context.SaveChanges();
  }
}
```

当运行应用程序来创建数据库时，只创建了一个表 Payments(如图 26-5 所示)。这个表定义了一个 Discriminator 列，将记录从表映射到相应的模型类型。

Payments
CreditcardNumber
PaymentId
Amount
Discriminator
Name

图 26-5

要只查询层次结构中的特定类型，可以使用 OfType 扩展方法。在下面的代码片段中，可以看到一个查询，该查询只返回 CreditcardPayment 类型的支付(代码文件 TPHWithConventions/Program.cs)：

```
private static void QuerySample()
{
  using (var context = new BankContext())
  {
    var creditcardPayments = context.Payments.OfType<CreditcardPayment>();
    foreach (var payment in creditcardPayments)
    {
      Console.WriteLine($"{payment.Name}, {payment.Amount}");
    }
  }
}
```

使用 OfType, EF Core 创建一个带有 WHERE 子句的查询，该子句只区分值为 CreditcardPayment 的记录：

```
SELECT [p].[PaymentId], [p].[Amount], [p].[Discriminator], [p].[Name],
  [p].[CreditcardNumber]
FROM [Payments] AS [p]
WHERE [p].[Discriminator] = N'CreditcardPayment'
```

当然，在这个场景中，还可以调用上下文属性 CreditcardPayments，得到相同的查询。

### 26.6.7 使用流利 API 的每个层次结构中的表

使用流利 API，定义层次结构可以有更多的控制。这样，Payment 类就从注释中剥离出来，它现在是一个抽象类型(代码文件 TPHWithFluentAPI/Payment.cs)：

```
public abstract class Payment
{
  public int PaymentId { get; set; }
  public string Name { get; set; }
  public decimal Amount { get; set; }
}
```

CreditcardPayment 和 Payment 类与前面的示例相同，因此这里不重复。但上下文是不同的。鉴别器的新名称是 Type。这应该是 Payments 表中的一列，但不应该显示在 Payment 类型中。应该使用字符串 Cash 和 Creditcard 来区分模型类型。对于所有字符串，定义了 ColumnNames 和 ColumnValues 类(代码文件 TPHWithFluentAPI/BankContext.cs)：

```
public static class ColumnNames
{
  public const string Type = nameof(Type);
}

public static class ColumnValues
{
  public const string Cash = nameof(Cash);
```

```
  public const string Creditcard = nameof(Creditcard);
}
```

这次，上下文为所有不同的 Payment 类型定义了一个属性 Payments。当然，也可以有专门的属性，但是在前面的示例中，这是必需的。对于 Name 属性和 Amount 属性的 Money 类型，需要的模式信息现在是在方法 onModelCreating 中指定的，而不是使用注释指定。使用 HasDiscriminator 方法指定 TPH 层次结构。鉴别器的名称是 Type，它也指定为一个阴影属性。派生类型的差异用 HasValue 方法指定。HasValue 是 DiscriminatorBuilder 的一个方法，它是从 HasDiscriminator 方法返回的。

```
public class BankContext : DbContext
{
  //...
  protected override void OnModelCreating(ModelBuilder modelBuilder)
  {
    modelBuilder.Entity<Payment>().Property(p => p.Name).IsRequired();
    modelBuilder.Entity<Payment>().Property(p => p.Amount)
      .HasColumnType("Money");

    // shadow property for the discriminator
    modelBuilder.Entity<Payment>().Property<string>(ColumnNames.Type);
    modelBuilder.Entity<Payment>()
      .HasDiscriminator<string>(ColumnNames.Type)
      .HasValue<CashPayment>(ColumnValues.Cash)
      .HasValue<CreditcardPayment>(ColumnValues.Creditcard);
  }

  public DbSet<Payment> Payments { get; set; }
}
```

创建的数据库与以前类似，只是表 Payments 现在定义了 Type 列，而不是 Discriminator 列，就像创建模型时指定的那样。询问信用卡号码的新查询会筛选 Type 列：

```
SELECT [p].[PaymentId], [p].[Amount], [p].[Name], [p].[Type],
  [p].[CreditcardNumber]
FROM [Payments] AS [p]
WHERE [p].[Type] = N'Creditcard'
```

### 26.6.8　表的拆分

表的拆分是 EF Core 2.0 的另一个新特性。通过表的拆分，可以将数据库表拆分为多个实体类型。使用表拆分特性，属于同一个表的每个类都需要一个一对一关系，并定义自己的主键。但是，因为它们共享同一个表，所以也共享相同的主键。

下面是一个 Menu 类的示例，它表示关于午餐菜单的信息，MenuDetails 包含厨房的信息。Menu 类为菜单定义了一些属性，包括 Details 属性。Details 属性将关系映射到 MenuDetails 类(代码文件 TableSplitting/Menu.cs)：

```
public class Menu
{
  public int MenuId { get; set; }
  public string Title { get; set; }
  public string Subtitle { get; set; }
  public decimal Price { get; set; }
  public MenuDetails Details { get; set; }
}
```

MenuDetails 类看起来会映射到它自己的表(带有主键)，并映射到具有 Menu 属性的 Menu 类(代码文件 TableSplitting/MenuDetails.cs)：

```
public class MenuDetails
{
  public int MenuDetailsId { get; set; }
  public string KitchenInfo { get; set; }
  public int MenusSold { get; set; }
  public Menu Menu { get; set; }
}
```

在上下文中，Menus 和 MenuDetails 是两个 DbSet 属性。在 OnModelCreating 方法中，Menu 类使用 HasOne 和 WithOne 配置为与 MenuDetails 的一对一关系。现在，应该注意 ToTable 方法的调用。如果没有这些代码行，

默认情况下，类 Menu 和 MenuDetails 将映射到两个不同的表。这里，传递给 ToTable 方法的参数指定了相同的表名。Menu 和 MenuDetails 都映射到相同的表 Menu。这就造成了表分割的差异(代码文件 TableSplitting/MenusContext.cs)：

```
public static class SchemaNames
{
  public const string Menus = nameof(Menus);
}

public class MenusContext : DbContext
{
  //...
  protected override void OnModelCreating(ModelBuilder modelBuilder)
  {
    modelBuilder.Entity<Menu>()
      .HasOne<MenuDetails>(m => m.Details)
      .WithOne(d => d.Menu)
      .HasForeignKey<MenuDetails>(d => d.MenuDetailsId);
    modelBuilder.Entity<Menu>().ToTable(SchemaNames.Menus);
    modelBuilder.Entity<MenuDetails>().ToTable(SchemaNames.Menus);
  }

  public DbSet<Menu> Menus { get; set; }
  public DbSet<MenuDetails> MenuDetails { get; set; }
}
```

验证表是如何在数据库中生成时，可以通过下面的 SQL 语句看到，Menu 表包含 Menu 和 MenuDetails 类的列，以及只用于 Menu 类的主键：

```
CREATE TABLE [dbo].[Menus](
  [MenuId] [int] IDENTITY(1,1) NOT NULL,
  [Price] [decimal](18, 2) NOT NULL,
  [Subtitle] [nvarchar](max) NULL,
  [Title] [nvarchar](max) NULL,
  [KitchenInfo] [nvarchar](max) NULL,
  [MenusSold] [int] NOT NULL,
CONSTRAINT [PK_Menus] PRIMARY KEY CLUSTERED
(
  [MenuId] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
  ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]
GO
```

**注意：**

实体分割是表分割的反过程，其中，一个实体被分割成多个表。这一功能尚未在 EF Core 2.0 中使用，但计划在 EF Core 的后续版本中使用。

## 26.6.9 拥有的实体

将表分割为多个实体类型的另一种方法是使用所谓“拥有的实体”的特性。拥有的实体不需要主键；它们可以是在正常实体中拥有的类型。拥有实体的实体类可以映射到单个表——使用表拆分特性——也可以映射到不同的表。当使用不同的表时，它们共享相同的主键。

下面看一个例子，它展示了这两种场景：使用拥有的实体和单个表，并将其映射到另一个表。

下面的代码片段显示了主要的实体类型 Person。这是带主键 PersonId 的拥有实体的所有者。该类型包含两个地址：PrivateAddress 和 CompanyAddress(代码文件 OwnedEntities/Person.cs)：

```
public class Person
{
  public int PersonId { get; set; }
  public string Name { get; set; }
  public Address PrivateAddress { get; set; }
  public Address CompanyAddress { get; set; }
}
```

Address 是一个拥有的实体——该类型没有它自己的主键。该类型有两个字符串属性，以及一个类型为

Location 的关系 Location。Location 是另一个拥有的实体(代码文件 OwnedEntities/Address.cs)：

```
public class Address
{
  public string LineOne { get; set; }
  public string LineTwo { get; set; }
  public Location Location { get; set; }
}
```

Location 只包含 Country 和 City 属性，作为一个拥有的实体，它也没有定义一个键(代码文件 OwnedEntities/Location.cs)：

```
public class Location
{
  public string Country { get; set; }
  public string City { get; set; }
}
```

最有趣的部分现在出现在上下文中，其中在 OnModelCreating 方法中定义了拥有的实体。为给 Person 类定制模型，OwnsOne 的第一次调用指定 Person 实体拥有从 CompanyAddress 属性(这是一种 Address 类型)引用的实体。对 OwnsOne 的第二个调用现在使用第一个 OwnsOne 调用(一个 ReferenceOwnershipBuilder)的返回类型调用 OwnsOne。这样，Address 就定义为拥有一个 Location。对于第二个调用，使用 OwnsOne 的另一个重载，允许进行一些定制。显示此自定义后，指定 City 和 Country 属性的列名与默认名称不同。CompanyAddress 定制的结果是，用于 CompanyAddress 和 Location 的列都包含在 People 表中，为 City 和 Country 属性提供定制的列名。使用下一个 OwnsOne 调用的定制定义了 PrivateAddress 属性的所有权。这一次，Address 类型映射到另一个表：映射到名为 Addr 的表。这个表还包含了 Location 类中的列，且带有默认列名 (代码文件 OwnedEntities/OwnedEntitiesContext.cs)：

```
public class OwnedEntitiesContext : DbContext
{
  //...
  protected override void OnModelCreating(ModelBuilder modelBuilder)
  {
    modelBuilder.Entity<Person>()
      .OwnsOne(p => p.CompanyAddress)
      .OwnsOne<Location>(a => a.Location, builder =>
      {
        builder.Property(p => p.City).HasColumnName("BusinessCity");
        builder.Property(p => p.Country).HasColumnName("BusinessCountry");
      });
    modelBuilder.Entity<Person>()
      .OwnsOne(p => p.PrivateAddress)
      .ToTable("Addr")
      .OwnsOne(a => a.Location);
  }

  public DbSet<Person> People { get; set; }
}
```

运行示例应用程序时，它将使用两个表创建数据库。第一个表是这里所示的 People 表。该表包含 Person 类的所有列，以及映射自 CompanyAddress 属性的 Address 和 Location 类。LineOne 和 LineTwo 具有属性名连接的默认命名，在分级属性名之间使用下划线：

```
CREATE TABLE [dbo].[People](
  [PersonId] [int] IDENTITY(1,1) NOT NULL,
  [Name] [nvarchar](max) NULL,
  [CompanyAddress_LineOne] [nvarchar](max) NULL,
  [CompanyAddress_LineTwo] [nvarchar](max) NULL,
  [BusinessCity] [nvarchar](max) NULL,
  [BusinessCountry] [nvarchar](max) NULL,
CONSTRAINT [PK_People] PRIMARY KEY CLUSTERED
(
  [PersonId] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
  ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]
GO
```

创建的第二个表(Addr)是由于 PrivateAddress 属性上的 ToTable 映射。这些列有默认命名，因为没有其他定

义。此表的键值与 People 表相同(PersonId)：

```
CREATE TABLE [dbo].[Addr](
  [PersonId] [int] NOT NULL,
  [LineOne] [nvarchar](max) NULL,
  [LineTwo] [nvarchar](max) NULL,
  [Location_City] [nvarchar](max) NULL,
  [Location_Country] [nvarchar](max) NULL,
CONSTRAINT [PK_Addr] PRIMARY KEY CLUSTERED
(
  [PersonId] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
  ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]
GO
```

# 26.7 保存数据

在使用模型和关系创建数据库之后，可以对其进行写入。26.2 节“EF Core 简介”展示了如何添加、更新和删除记录，但是现在深入了解如何影响这些记录并包括关系。

## 26.7.1 用关系添加对象

下面的代码片段写入一个关系：MenuCard 包含 Menu 对象。其中，实例化 MenuCard 和 Menu 对象，再指定双向关联。对于 Menu，将 MenuCard 属性分配给 MenuCard，对于 MenuCard，用 Menu 对象填充 Menus 属性。调用 MenuCards 属性的方法 Add，把 MenuCard 实例添加到上下文中。将对象添加到上下文时，默认情况下所有对象都添加到树中，并添加状态。不仅保存 MenuCard，还保存 Menu 对象。在上下文中调用 SaveChanged，会创建 4 个记录(代码文件 MenusSample/Program.cs)：

```
private static void AddRecords()
{
  //...
  using (var context = new MenusContext())
  {
    var soupCard = new MenuCard();
    Menu[] soups =
    {
      new Menu
      {
        Text = "Consommé Célestine (with shredded pancake)",
        Price = 4.8m,
        MenuCard = soupCard
      },
      new Menu
      {
        Text = "Baked Potato Soup",
        Price = 4.8m,
        MenuCard = soupCard
      },
      new Menu
      {
        Text = "Cheddar Broccoli Soup",
        Price = 4.8m,
        MenuCard = soupCard
      },
    };
    soupCard.Title = "Soups";
    soupCard.Menus.AddRange(soups);
    context.MenuCards.Add(soupCard);
    ShowState(context);
    int records = context.SaveChanges();
    Console.WriteLine($"{records} added");
    //...
  }
}
```

给上下文添加 4 个对象后调用的方法 ShowState 显示了所有与上下文相关的对象的状态。DbContext 类有一个相关的 ChangeTracker，使用 ChangeTracker 属性可以访问它。ChangeTracker 的 Entries 方法返回变更跟踪器

了解的所有对象。在 foreach 循环中，每个对象包括其状态都写入控制台(代码文件 MenusSample/Program.cs)：

```
public static void ShowState(MenusContext context)
{
  foreach (EntityEntry entry in context.ChangeTracker.Entries())
  {
    Console.WriteLine($"type: {entry.Entity.GetType().Name}, " +
      $"state: {entry.State}, {entry.Entity}");
  }
  Console.WriteLine();
}
```

运行应用程序，查看 4 个对象的 Added 状态：

```
type: MenuCard, state: Added, Soups
type: Menu, state: Added, Consommé Célestine (with shredded pancake)
type: Menu, state: Added, Baked Potato Soup
type: Menu, state: Added, Cheddar Broccoli Soup
```

因为这个状态，SaveChanges 方法创建 SQL Insert 语句，把每个对象写到数据库。

## 26.7.2　对象的跟踪

如前所述，上下文知道添加的对象。然而，上下文也需要了解变更。要了解变更，每个检索的对象就需要它在上下文中的状态。为了查看这个状态，下面创建两个不同的查询，但返回相同的对象。下面的代码片段定义了两个不同的查询，每个查询都用菜单返回相同的对象，因为它们都存储在数据库中。事实上，只有一个对象会物化，因为在第二个查询的结果中，返回的记录具有的主键值与从上下文中引用的对象相同。验证在返回相同的对象时，变量 m1 和 m2 的引用是否具有相同的结果(代码文件 MenusSample/Program.cs)：

```
private static void ObjectTracking()
{
  using (var context = new MenusContext())
  {
    var m1 = (from m in context.Menus
              where m.Text.StartsWith("Con")
              select m).FirstOrDefault();
    var m2 = (from m in context.Menus
              where m.Text.Contains("(")
              select m).FirstOrDefault();
    if (object.ReferenceEquals(m1, m2))
    {
      Console.WriteLine("the same object");
    }
    else
    {
      Console.WriteLine("not the same");
    }
    ShowState(context);
  }
}
```

第一个 LINQ 查询得到一个带 LIKE 比较的 SQL SELECT 语句，来比较以 Con 开头的字符串：

```
SELECT TOP(1) [m].[MenuId], [m].[MenuCardId], [m].[Price], [m].[Text]
FROM [mc].[Menus] AS [m]
WHERE [m].[Text] LIKE 'Con' + '%'
```

在第二个 LINQ 查询中，也需要咨询数据库。其中 LIKE 用于比较文字中间的“(”：

```
SELECT TOP(1) [m].[MenuId], [m].[MenuCardId], [m].[Price], [m].[Text]
FROM [mc].[Menus] AS [m]
WHERE [m].[Text] LIKE ('%' + '(') + '%'
```

运行应用程序时，同一对象写入控制台，只有一个对象用 ChangeTracker 保存。状态是 Unchanged：

```
the same object
type: Menu, state: Unchanged, Consommé Célestine (with shredded pancake)
```

为了不跟踪在数据库中运行查询的对象，可以通过 DbSet 调用 AsNoTracking 方法：

```
var m1 = (from m in context.Menus.AsNoTracking()
          where m.Text.StartsWith("Con")
          select m).FirstOrDefault();
```

可以把 ChangeTracker 的默认跟踪行为配置为 QueryTrackingBehavior.NoTracking：

```
using (var context = new MenusContext())
{
  context.ChangeTracker.QueryTrackingBehavior =
    QueryTrackingBehavior.NoTracking;
  //...
}
```

对于更全局的配置，跟踪行为也可以用 SQL Server 配置来配置：

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
  base.OnConfiguring(optionsBuilder);
  optionsBuilder.UseSqlServer(ConnectionString)
    .UseQueryTrackingBehavior(QueryTrackingBehavior.NoTracking);
}
```

有了这样的配置，给数据库建立两个查询，物化两个对象，状态信息是空的。

**注意：**
当上下文只用于读取记录时，可以使用 NoTracking 配置，但无法修改。这减少了上下文的开销，因为不保存状态信息。

## 26.7.3 更新对象

跟踪对象时，对象可以轻松地更新，如下面的代码片段所示。首先，检索 Menu 对象。使用这个被跟踪的对象，修改价格，再把变更写入数据库。在所有的变更之间，将状态信息写入控制台(代码文件 MenusSample/Program.cs)：

```
private static void UpdateRecords()
{
  using (var context = new MenusContext())
  {
    Menu menu = context.Menus
      .Skip(1)
      .FirstOrDefault();
    ShowState(context);
    menu.Price += 0.2m;
    ShowState(context);
    int records = context.SaveChanges();
    Console.WriteLine($"{records} updated");
    ShowState(context);
  }
}
```

运行应用程序时，可以看到，加载记录后，对象的状态是 Unchanged，修改属性值后，对象的状态是 Modified，保存完成后，对象的状态是 Unchanged：

```
type: Menu, state: Unchanged, Baked Potato Soup
type: Menu, state: Modified, Baked Potato Soup
1 updated
type: Menu, state: Unchanged, Baked Potato Soup
```

访问更改跟踪器中的条目时，默认情况下会自动检测到变更。要配置这个，应设置 ChangeTracker 的 AutoDetectChangesEnabled 属性。为了手动检查更改是否已经完成，调用 DetectChanges 方法。调用 SaveChangesAsync 后，状态改回 Unchanged。调用 AcceptAllChanges 方法可以手动完成这个操作。

## 26.7.4 更新未跟踪的对象

DB 上下文通常非常短寿。使用 EF Core 与 ASP.NET Core MVC ，通过一个 HTTP 请求创建一个对象上下文，来检索对象。从客户端接收一个更新时，对象必须再在服务器上创建。这个对象与对象的上下文相关联。为了在数据库中更新它，对象需要与 DB 上下文相关联，修改状态，创建 INSERT、UPDATE 或 DELETE 语句。

这样的情景用下一个代码片段模拟。本地函数 GetMenu 返回一个脱离上下文的 Menu 对象；上下文在该本地函数的最后销毁。GetMenu 方法由 ChangeUntracked 方法调用。这个方法修改不与任何上下文相关的 Menu

对象。改变后，Menu 对象传递到方法 UpdateUntracked，保存到数据库中(代码文件 MenusSample/Program.cs)：

```
private static void ChangeUntracked()
{
  Menu GetMenu()
  {
    using (var context = new MenusContext())
    {
      Menu menu = context.Menus
        .Skip(2)
        .FirstOrDefault();
      return menu;
    }
  }
  Menu m = GetMenu();
  m.Price += 0.7m;
  UpdateUntracked(m);
}
```

UpdateUntracked 方法接收已更新的对象，需要把它与上下文关联起来。对象与上下文关联起来的一个方法是调用 DbSet 的 Attach 方法，并根据需要设置状态。Update 方法用一个调用完成这两个操作：关联对象，把状态设置为 Modified (代码文件 MenusSample/Program.cs)：

```
private static void UpdateUntracked(Menu m)
{
  using (var context = new MenusContext())
  {
    ShowState(context);
    // EntityEntry<Menu> entry = context.Menus.Attach(m);
    // entry.State = EntityState.Modified;
    context.Menus.Update(m);
    ShowState(context);
    context.SaveChanges();
  }
}
```

通过 ChangeUntracked 方法运行应用程序时，可以看到状态的修改。对象起初没有被跟踪，但是，因为显式地更新了状态，所以可以看到 Modified 状态：

```
type: Menu, state: Modified, Cheddar Broccoli Soup
```

## 26.7.5　批处理

对象映射工具不支持所有场景。例如，如果城市的邮政编码更改为新代码，并且希望把所有客户的旧邮政编码更新为新代码，最好调用一个 SQL UPDATE 语句来更新所有这些记录。使用 EF Core，为每个客户生成更新语句。

但是，EF Core 对于通过一个 SaveChanges 调用发送一系列单独的 SQL 语句并没有那么糟糕。EF Core 支持批处理。SaveChanges 向 SQL Server 发送一个命令，其中仅用一条语句执行多个插入或更新操作。可以控制批处理的大小——例如，当配置 SQL Server 时，通过调用值为 1 的 MaxBatchSize()来禁用批处理：

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
  base.OnConfiguring(optionsBuilder);
  optionsBuilder.UseSqlServer(ConnectionString,
    options => options.MaxBatchSize(1));
}
```

下面的代码片段创建了 100 个菜单对象，并将它们作为添加对象添加到上下文中，用 SaveChanges 方法将其写入数据库(代码文件 MenusSample/Program.cs)：

```
private static void AddHundredRecords()
{
  Console.WriteLine(nameof(AddHundredRecords));
  using (var context = new MenusContext())
  {
    var card = context.MenuCards.FirstOrDefault();
    if (card != null)
    {
      var menus = Enumerable.Range(1, 100).Select(x => new Menu
```

```
        {
          MenuCard = card,
          Text = $"menu {x}",
          Price = 9.9m
        });
        context.Menus.AddRange(menus);
        Stopwatch stopwatch = Stopwatch.StartNew();
        int records = context.SaveChanges();
        stopwatch.Stop();
        Console.WriteLine($"{records} records added after " +
          $"{stopwatch.EllapsedMilliseconds} milliseconds");
      }
    }
    Console.WriteLine();
  }
```

启用批处理，运行应用程序时，EF Core 提供程序创建一个 TABLE MERGE 语句，其中使用一条语句写入 100 条新菜单记录。将批处理的大小更改为 1100 个 INSERT 语句，发送到数据库。数据库在同一个系统上运行时，启用了批处理的时间间隔是 56 毫秒，而没有批处理的时间间隔是 105 毫秒。如果数据库在不同系统上运行，差异会更大。

# 26.8 冲突的处理

如果多个用户修改同一个记录，然后保存状态，会发生什么？最后谁的变更会保存下来?

如果访问同一个数据库的多个用户处理不同的记录，就没有冲突。所有用户都可以保存他们的数据，而不干扰其他用户编辑的数据。但是，如果多个用户处理同一记录，就需要考虑如何解决冲突。有不同的方法来处理冲突。最简单的一个方法是最后一个用户获胜。最后保存数据的用户覆盖以前用户执行的变更。

EF Core 还提供了一种方式，使第一个保存数据的用户获胜。采用这一选项，保存记录时，需要验证最初读取的数据是否仍在数据库中。如果是，就继续保存数据，因为读写操作之间没有发生变化。然而，如果数据发生了变化，就需要解决冲突。

下面看看这些不同的选项。

## 26.8.1 最后一个更改获胜

默认情况是，最后一个保存的更改获胜。为了查看对数据库的多个访问，扩展 BooksSample 应用程序。

为了简单地模拟两个用户，方法 ConflictHandling 调用两次方法 PrepareUpdate，对两个引用相同记录的 Book 对象进行不同的改变，并调用 Update 方法两次。最后，把书的 ID 传递给 CheckUpdate 方法，它显示了数据库中书的实际状态(代码文件 BooksSample/Program.cs)：

```
public static void ConflictHandling()
{
  // user 1
  var tuple1 = await PrepareUpdate();
  tuple1.book.Title = "updated from user 1";

  // user 2
  var tuple2 = await PrepareUpdate();
  tuple2.book.Title = "updated from user 2";

  // user 1
  Update(tuple1.context, tuple1.book, "user 1");

  // user 2
  UpdateAsync(tuple2.context, tuple2.book, "user 2");

  tuple1.context.Dispose();
  tuple2.context.Dispose();

  CheckUpdate(tuple1.book.BookId);
}
```

PrepareUpdate 方法打开一个 BookContext，并在元组中返回上下文和图书。记住，该方法调用两次，返回

与不同 context 对象相关的不同 Book 对象(代码文件 ConflictHandlingSample/Program.cs)：

```
private static (BooksContext context, Book book) PrepareUpdate()
{
  var context = new BooksContext();
  Book book = await context.Books
    .Where(b => b.Title == BookTitle)
    .FirstOrDefault();
  return (context, book);
}
```

**注意：**
*元组在第 13 章介绍。*

Update 方法接收打开的 BooksContext 与更新的 Book 对象，把这本书保存到数据库中。记住，该方法调用两次(代码文件 BooksSample/Program.cs)：

```
private static void Update(BooksContext context, Book book, string user)
{
  int records = context.SaveChanges();
  Console.WriteLine($"{user}: {records} record updated from {user}");
}
```

CheckUpdate 方法把指定 id 的图书写到控制台(代码文件 BooksSample/Program.cs)：

```
private static void CheckUpdate(int id)
{
  using (var context = new BooksContext())
  {
    Book book = context.Books.Find(id);
    Console.WriteLine($"updated: {book.Title}");
  }
}
```

运行应用程序时，会发生什么？第一个更新会成功，第二个更新也会成功。更新一条记录时，不验证读取记录后是否发生变化，这个示例应用程序就是这样。第二个更新会覆盖第一个更新的数据，如应用程序的输出所示：

```
user 1: 1 record updated from user 1
user 2: 1 record updated from user 2
this is the updated state: updated from user 2
```

## 26.8.2　第一个更改获胜

如果需要不同的行为，如第一个用户的更改保存到记录中，就需要做一些改变。示例项目 ConflictHandlingSample 像以前一样使用 Book 和 BookContext 对象，但它处理第一个更改获胜的场景。

为了解决冲突，需要指定属性，如果在读取和更新之间发生了变化，就应使用并发性令牌验证该属性。基于指定的属性，修改 SQL UPDATE 语句，不仅验证主键，还验证用并发性令牌标记的所有属性。给实体类型添加许多并发性令牌，会在 UPDATE 语句中创建一个巨大的 WHERE 子句，这不是非常有效。相反，可以添加一个属性，在 SQL Server 中用每个 UPDATE 语句更新——这就是 Book 类完成的工作。属性 TimeStamp 在 SQL Server 中定义为 timeStamp(代码文件 ConflictHandlingSample/Book.cs)：

```
public class Book
{
  public int BookId { get; set; }
  public string Title { get; set; }
  public string Publisher { get; set; }
  public byte[] TimeStamp { get; set; }
}
```

在 SQL Server 中将 TimeStamp 属性定义为 timestamp 类型，要使用 Fluent API。SQL 数据类型使用 HasColumnType 方法定义。方法 ValueGeneratedOnAddOrUpdate 通知上下文，在每一个 SQL INSERT 或 UPDATE 语句中，可以改变 TimeStamp 属性，这些操作后，它需要用上下文设置。IsConcurrencyToken 方法将这个属性标记为必要，检查它在读取操作完成后是否没有改变(代码文件 ConflictHandlingSample/BooksContext.cs)：

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
  base.OnModelCreating(modelBuilder);
  var book = modelBuilder.Entity<Book>();
  book.HasKey(p => p.BookId);
  book.Property(p => p.Title).HasMaxLength(120).IsRequired();
  book.Property(p => p.Publisher).HasMaxLength(50);
  book.Property(p => p.TimeStamp)
    .HasColumnType("timestamp")
    .ValueGeneratedOnAddOrUpdate()
    .IsConcurrencyToken();
}
```

**注意：**

不使用 IsConcurrencyToken 方法与 Fluent API，也可以给应检查并发性的属性应用 ConcurrencyCheck 特性。

检查冲突处理的过程类似于之前的操作。用户 1 和用户 2 都调用 PrepareUpdate 方法，改变了书名，并调用 Update 方法修改数据库(代码文件 ConflictHandlingSample/Program.cs)：

```
public static void ConflictHandling()
{
  // user 1
  var tuple1 = PrepareUpdate();
  tuple1.book.Title = "user 1 wins";

  // user 2
  var tuple2 = await PrepareUpdate();
  tuple2.book.Title = "user 2 wins";

  // user 1
  Update(tuple1.context, tuple1.book);
  // user 2
  Update(tuple2.context, tuple2.book);
  tuple1.context.Dispose();
  tuple2.context.Dispose();
  CheckUpdate(context1.book.BookId);
}
```

这里不重复 PrepareUpdate 方法，因为该方法的实现方式与前面的示例相同。Update 方法则截然不同。为了查看更新前和更新后不同的时间戳，实现字节数组的自定义扩展方法 StringOutput，将字节数组以可读的形式写到控制台。接着，调用 ShowChanges 辅助方法，显示对 Book 对象的更改。调用 SaveChanges 方法，把所有更新写到数据库中。如果更新失败，并抛出 DbUpdateConcurrencyException 异常，就把失败信息写入控制台(代码文件 ConflictHandlingSample/Program.cs)：

```
private static Update(BooksContext context, Book book, string user)
{
  try
  {
    Console.WriteLine($"{user}: updating id {book.BookId}, " +
      $"timestamp: {book.TimeStamp.StringOutput()}");
    ShowChanges(book.BookId, context.Entry(book));
    int records = await context.SaveChangesAsync();
    Console.WriteLine($»{user}: updated {book.TimeStamp.StringOutput()}»);
    Console.WriteLine($"{user}: {records} record(s) updated while updating " +
      $"{book.Title}");
  }
  catch (DbUpdateConcurrencyException ex)
  {
    Console.WriteLine($"{user}: update failed with {book.Title}");
    Console.WriteLine($"error: {ex.Message}");
    foreach (var entry in ex.Entries)
    {
      if (entry.Entity is Book b)
      {
        Console.WriteLine($"{b.Title} {b.TimeStamp.StringOutput()}");
        ShowChanges(book.BookId, context.Entry(book));
      }
    }
  }
}
```

对于与上下文相关的对象，使用 PropertyEntry 对象可以访问原始值和当前值。从数据库中读取对象时获取

的原始值，可以用 OriginalValue 属性访问，其当前值可以用 CurrentValue 属性访问。在 ShowChanges 和 ShowChange 方法中，PropertyEntry 对象可以用 EntityEntry 的属性方法访问，如下所示(代码文件 ConflictHandlingSample/Program.cs)：

```
private static void ShowChanges(int id, EntityEntry entity)
{
  void ShowChange(PropertyEntry propertyEntry) =>
    Console.WriteLine($"id: {id}, current: {propertyEntry.CurrentValue}, " +
      $"original: {propertyEntry.OriginalValue}, " +
      $"modified: {propertyEntry.IsModified}");

  ShowChange(entity.Property("Title"));
  ShowChange(entity.Property("Publisher"));
}
```

为了转换 SQL Server 中更新的 TimeStamp 属性的字节数组，以可视化输出，定义了扩展方法 StringOutput(代码文件 ConflictHandlingSample/ByteArrayExtensions.cs)：

```
static class ByteArrayExtension
{
  public static string StringOutput(this byte[] data)
  {
    var sb = new StringBuilder();
    foreach (byte b in data)
    {
      sb.Append($"{b}.");
    }
    return sb.ToString();
  }
}
```

当运行应用程序时，可以看到如下输出。时间戳值和图书 ID 在每次运行时都不同。第一个用户把书的原标题 sample book 更新为新标题 user 1 wins。IsModified 属性给 Title 属性返回 true，但给 Publisher 属性返回 false。因为只有标题改变了。原来的时间戳以 1.1.209 结尾；更新到数据库中后，时间戳改为 1.17.114。与此同时，用户 2 打开相同的记录；该书的时间戳仍然是 1.1.209。用户 2 更新该书，但这里更新失败了，因为该书的时间戳不匹配数据库中的时间戳。这里会抛出一个 DbUpdateConcurrencyException 异常。在异常处理程序中，异常的原因写入控制台，如程序的输出所示：

```
user 1: updating id 1, timestamp 0.0.0.0.0.0.7.209.
id: 1, current: user 1 wins, original: sample book, modified: True
id: 1, current: Sample, original: Sample, modified: False
user 1: updated 0.0.0.0.0.0.7.210.
user 1: 1 record(s) updated while updating user 1 wins
user 2: updating id 1, timestamp 0.0.0.0.0.0.7.209.
id: 1, current: user 2 wins, original: sample book, modified: True
id: 1, current: Sample, original: Sample, modified: False
user 2 update failed with user 2 wins
user 2 error: Database operation expected to affect 1 row(s) but actually
affected 0 row(s). Data may have been modified or deleted since entities were loaded.
See http://go.microsoft.com/fwlink/?LinkId=527962 for information on
understanding and handling optimistic concurrency exceptions.
user 2 wins 0.0.0.0.0.0.7.209.
id: 1, current: user 2 wins, original: sample book, modified: True
id: 1, current: Sample, original: Sample, modified: False
this is the updated state: user 1 wins
```

当使用并发性令牌和处理 DbConcurrencyException 时，可以根据需要处理并发冲突。例如，可以自动解决并发问题。如果改变了不同的属性，可以检索更改的记录并合并更改。如果改变的属性是一个数字，要执行一些计算，例如点系统，就可以在两个更新中递增或递减值，如果达到极限，就抛出一个异常。也可以给用户提供数据库中目前的信息，询问他要进行什么修改，要求用户解决并发性问题。不要要求用户提供太多的信息。用户可能只是想摆脱这个很少显示的对话框，这意味着他可能会单击 OK 或 Cancel，而不阅读其内容。对于罕见的冲突，也可以编写日志，通知系统管理员，需要解决一个问题。

# 26.9 上下文池

在 EF Core 2.0 中，可以把上下文放在池中，以提高性能。连接已经放在池中好长时间了。应在需要连接之前打开它们，使用之后立即关闭它们。在 EF Core 中，这个行为已经在框架中实现了。关闭连接时，对数据库服务器的连接并没有真正关闭，而是把连接返回连接池，以便在打开下一个连接时重用。连接池用连接字符串配置。

DB 上下文的用法规则与连接类似。它们也应在需要之前创建，在使用之后立即关闭（销毁)。其开销没有我们想象的那么大。模型不是在调用每个新上下文时初始化，而是重用模型。在 Entity Framework 和 XML 文件映射中，创建上下文的开销比目前的 EF Core 大许多。但是，如果上下文比较大，创建它的开销仍可能比较可观。此时就可以使用上下文池来提高性能。

为了使用上下文池，必须使用依赖注入。要激活上下文池，只需要把 EF Core 注册从 AddDbContext 改为 AddDbContextPool。这样，注入的上下文（示例代码中是 BooksContext）就可以从上下文池中检索出来：

```
var services = new ServiceCollection();
services.AddTransient<BooksController>();
services.AddTransient<BooksService>();
services.AddEntityFrameworkSqlServer();
services.AddDbContextPool<BooksContext>(options =>
  options.UseSqlServer(ConnectionString));
services.AddLogging();

Container = services.BuildServiceProvider();
```

# 26.10 使用事务

第 25 章介绍了使用事务编程的内容。每次使用 Entity Framework 访问数据库时，都涉及事务。可以隐式地使用事务或根据需要使用配置显式地创建它们。此节使用的示例项目以两种方式展示事务。这里，Menu、MenuCard 和 MenuContext 类的用法与前面的 MenusSample 项目相同。

## 26.10.1 使用隐式的事务

SaveChanges 方法的调用会自动解析为一个事务。如果需要进行的一部分变更失败，例如，因为数据库约束，就回滚所有已经完成的更改。下面的代码片段演示了这一点。其中，第一个 Menu (m1)用有效的数据创建。对现有 MenuCard 的引用是通过提供 MenuCardId 完成的。更新成功后，Menu m1 的 MenuCard 属性自动填充。然而，所创建的第二个菜单 mInvalid，因为提供的 MenuCardId 高于数据库中可用的最高 ID，所以引用了无效的菜单卡。因为 MenuCard 和 Menu 之间定义了外键关系，所以添加这个对象会失败(代码文件 TransactionsSample/Program.cs)：

```
private static void AddTwoRecordsWithOneTx()
{
  Console.WriteLine(nameof(AddTwoRecordsWithOneTx));
  try
  {
    using (var context = new MenusContext())
    {
      var card = context.MenuCards.First();
      var m1 = new Menu
      {
        MenuCardId = card.MenuCardId,
        Text = "added",
        Price = 99.99m
      };

      int hightestCardId = context.MenuCards.Max(c => c.MenuCardId);
      var mInvalid = new Menu
      {
        MenuCardId = ++hightestCardId,
        Text = "invalid",
        Price = 999.99m
```

```
      };
      context.Menus.AddRange(m1, mInvalid);
      int records = context.SaveChanges();
      Console.WriteLine($"{records} records added");
    }
  }
  catch (DbUpdateException ex)
  {
    Console.WriteLine($"{ex.Message}");
    Console.WriteLine($"{ex?.InnerException.Message}");
  }
  Console.WriteLine();
}
```

在调用 AddTwoRecordsWithOneTx 方法运行应用程序之后，可以验证数据库的内容，确定没有添加一个记录。异常消息以及内部异常的消息给出了细节：

```
AddTwoRecordsWithOneTx
trying to add one invalid record to the database, this should fail...
An error occurred while updating the entries. See the inner exception for details.
The MERGE statement conflicted with the FOREIGN KEY constraint
"FK_Menus_MenuCards_MenuCardId".
The conflict occurred in database "ProCSharpMenuCards",
table "mc.MenuCards", column 'MenuCardId'.
The statement has been terminated.
```

如果第一条记录写入数据库应该是成功的，即使第二条记录写入失败，也需要多次调用 SaveChanges 方法，如下面的代码片段所示。在 AddTwoRecordsWithTwoTx 方法中，第一次调用 SaveChanges 插入了 m1 菜单对象，而第二次调用试图插入 mInvalidMenu 对象(代码文件 TransactionsSample/Program.cs)：

```
private static void AddTwoRecordsWithTwoTx()
{
  Console.WriteLine(nameof(AddTwoRecordsWithTwoTx));
  try
  {
    using (var context = new MenusContext())
    {
      var card = context.MenuCards.First();
      var m1 = new Menu
      {
        MenuCardId = card.MenuCardId,
        Text = "added",
        Price = 99.99m
      };
      context.Menus.Add(m1);
      int records = context.SaveChanges();
      Console.WriteLine($"{records} records added");

      int hightestCardId = context.MenuCards.Max(c => c.MenuCardId);
      var mInvalid = new Menu
      {
        MenuCardId = ++hightestCardId,
        Text = "invalid",
        Price = 999.99m
      };
      context.Menus.Add(mInvalid);
      records = context.SaveChanges();
      Console.WriteLine($"{records} records added");
    }
  }
  catch (DbUpdateException ex)
  {
    Console.WriteLine($"{ex.Message}");
    Console.WriteLine($"{ex?.InnerException.Message}");
  }
  Console.WriteLine();
}
```

运行应用程序，添加第一个 INSERT 语句成功，当然第二个语句的结果是 DbUpdateException。可以验证数据库，这次添加一个记录：

```
AddTwoRecordsWithTwoTx
adding two records with two transactions to the database.
One record should be written, the other not....
1 records added
```

```
An error occurred while updating the entries. See the inner exception for details.
The INSERT statement conflicted with the FOREIGN KEY constraint "FK_Menus_MenuCards_MenuCardId".
The conflict occurred in database "ProCSharpMenuCards",
table "mc.MenuCards", column 'MenuCardId'.
The statement has been terminated.
```

## 26.10.2 创建显式的事务

除了使用隐式创建的事务之外，也可以显式地创建它们。其优势是如果一些业务逻辑失败，也可以选择回滚，还可以在一个事务中结合多个 SaveChanges 调用。为了开始一个与 DbContext 派生类相关的事务，需要调用 DatabaseFacade 类中从 Database 属性返回的 BeginTransaction 方法。返回的事务实现了 IDbContextTransaction 接口。与 DbContext 相关的 SQL 语句通过事务建立起来。为了提交或回滚，必须显式地调用 Commit 或 Rollback 方法。在示例代码中，当达到 DbContext 作用域的末尾时，Commit 完成，在发生异常的情况下回滚(代码文件 TransactionsSample/Program.cs)：

```
private static void TwoSaveChangesWithOneTx()
{
  Console.WriteLine(nameof(TwoSaveChangesWithOneTxAsync));
  IDbContextTransaction tx = null;
  try
  {
    using (var context = new MenusContext())
    using (tx = await context.Database.BeginTransaction())
    {
      var card = context.MenuCards.First();
      var m1 = new Menu
      {
        MenuCardId = card.MenuCardId,
        Text = "added with explicit tx",
        Price = 99.99m
      };
      context.Menus.Add(m1);
      int records = await context.SaveChanges();
      Console.WriteLine($"{records} records added");

      int hightestCardId = context.MenuCards.Max(c => c.MenuCardId);
      var mInvalid = new Menu
      {
        MenuCardId = ++hightestCardId,
        Text = "invalid",
        Price = 999.99m
      };
      context.Menus.Add(mInvalid);
      records = await context.SaveChanges();

      Console.WriteLine($"{records} records added");
      tx.Commit();
    }
  }
  catch (DbUpdateException ex)
  {
    Console.WriteLine($"{ex.Message}");
    Console.WriteLine($"{ex?.InnerException.Message}");
    Console.WriteLine("rolling back…");
    tx.Rollback();
  }
  Console.WriteLine();
}
```

当运行应用程序时可以看到，没有添加记录，但多次调用了 SaveChanges 方法。SaveChanges 的第一次返回列出了要添加的一个记录，但基于后面的 Rollback，删除了这个记录。根据隔离级别的设置，回滚完成之前，更新的记录只能在事务内可见，但在事务外部不可见。

```
TwoSaveChangesWithOneTx
using one explicit transaction, writing should roll back...
1 records added
An error occurred while updating the entries. See the inner exception for details.
The INSERT statement conflicted with the FOREIGN KEY constraint
"FK_Menus_MenuCards_MenuCardId".
The conflict occurred in database "ProCSharpMenuCards",
table "mc.MenuCards", column 'MenuCardId'.
```

```
The statement has been terminated.
rolling back...
```

**注意：**

使用 BeginTransaction 方法，也可以给隔离级别提供一个值，指定数据库中所需的隔离要求和锁。隔离级别参见第 25 章。

# 26.11　迁移

可以在现有的数据库中使用 EF Core(称为“数据库优先”)。在许多使用 EF Core 的场景中，数据库已经存在。数据库的更新独立于应用程序，并且在数据库更改完成后更新应用程序。在这种情况下，EF Core 迁移没有多大帮助。

如果使用应用程序创建数据库，EF Core 迁移将非常有用。更改代码模型时，可以自动更新数据库。如果客户都有自己的数据库，并且使用应用程序的更新版本更改数据库模式，那么使用旧的应用程序版本更新客户可能是一个挑战。EF Core 迁移可以解决这个问题：通过迁移，可以轻松地从版本 x 升级到版本 y。数据库的当前版本是从数据库中读取的，而迁移则包含了升级到最新版本的每一步所需的信息。还可以升级或降级到特定的版本。

有不同的选项来升级数据库。使用升级命令可以直接从应用程序迁移。还可以使用 dotnet 命令从命令行上更新数据库。另一个选项是创建一个 SQL Server 脚本，数据库管理员可以使用该脚本更新数据库。

显示迁移的示例应用程序存在于.NET 标准库和.NET Core Web 应用程序中。通常，数据访问代码是在库中实现的，需要一些额外的命令行选项来处理这个问题，这就是为什么在这样的场景中演示迁移的原因。

## 26.11.1　准备项目文件

下面从.NET 标准 2.0 库开始。这个库包含定义模型的 Menu 和 MenuCard 类、实现接口 IEntityTypeConfiguration 来配置对应模型类型的映射的 MenuConfiguration 和 MenuCardConfiguration 类，以及上下文类 MenusContext。

项目文件需要准备的不仅仅是引用 NuGet 包 Microsoft.EntityFrameworkCore 和 Microsoft.EntityFrameworkCore.SqlServer，还需要 EF Core 工具扩展与 dotnet 命令行，这是一个引用 Microsoft.EntityFrameworkCore.Tools.Dotnet 的工具 (项目文件 MigrationsLib/MigrationsLib.csproj)：

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.EntityFrameworkCore"
      Version="2.0.0" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer"
      Version="2.0.0" />
  </ItemGroup>

  <ItemGroup>
    <DotNetCliToolReference
      Include="Microsoft.EntityFrameworkCore.Tools.Dotnet"
      Version="2.0.0" />
  </ItemGroup>

</Project>
```

为了使用 Web 应用程序中的上下文类，实现了 MenusContext，以使用依赖注入和需要 DbContextOptions 的构造函数(代码文件 MigrationsLib/MenusContext.cs)：

```
public class MenusContext : DbContext
{
  public MenusContext(DbContextOptions<MenusContext> options):
```

```
    base(options) { }

  public DbSet<Menu> Menus { get; set; }
  public DbSet<MenuCard> MenuCards { get; set; }

  protected override void OnModelCreating(ModelBuilder modelBuilder)
  {
    base.OnModelCreating(modelBuilder);
    modelBuilder.HasDefaultSchema("mc");

    modelBuilder.ApplyConfiguration(new MenuCardConfiguration());
    modelBuilder.ApplyConfiguration(new MenuConfiguration());
  }
}
```

### 26.11.2 利用 ASP.NET Core MVC 托管应用程序

在新的 ASP.NET Core MVC Web 应用程序中，使用 Microsoft.Extensions.DependencyInjection 的依赖注入已经构建到模板中。要启用迁移，只需要使用扩展方法 AddDbContext 添加 EF Core DB 上下文，并使用 UseSqlServer 配置 SQL Server。需要在配置文件中配置到数据库的连接字符串。这样，就可以使用命令行进行迁移(代码文件 MigrationsWebApp/Startup.cs)：

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddMvc();
  services.AddDbContext<MenusContext>(options =>
    options.UseSqlServer(
      Configuration.GetConnectionString("MenusConnection")));
}
```

**注意：**

ASP.NET Core MVC 详见第 31 章。在此之前，应该阅读第 20 章和第 30 章。

### 26.11.3 托管.NET Core 控制台应用程序

如果 EF Core DB 上下文没有使用依赖注入，就可以使用配置了工具的上下文类。在这种情况下，可以使用上下文的 OnConfiguration 方法检索连接字符串。使用依赖项注入，连接字符串从外部注入。ASP .NET Core 在访问 Web 应用程序的 Main()方法，以通过依赖注入容器获取连接字符串方面有工具的特殊支持。控制台应用程序(以及 UWP、WPF 和 Xamarin 应用程序)中的 Main()方法看起来有所不同。在这里，需要实现一个工厂类，通过实现接口 IDesignTimeDbContextFactory 返回 DB 上下文。当访问程序集时，在.NET Core 工具中自动检测到实现该接口的类。这个接口定义的 CreateDbContext 方法需要返回一个已配置的上下文(代码文件 MigrationsConsoleApp/MenusContextFactory.cs)：

```
public class MenusContextFactory : IDesignTimeDbContextFactory<MenusContext>
{
  private const string ConnectionString =
  @"server=(localdb)\mssqllocaldb;database=ProCSharpMigrations;" +
    "trusted_connection=true";

  public MenusContext CreateDbContext(string[] args)
  {
    var optionsBuilder = new DbContextOptionsBuilder<MenusContext>();
    optionsBuilder.UseSqlServer(ConnectionString);
    return new MenusContext(optionsBuilder.Options);
  }
}
```

.NET Core 工具使用的控制台应用程序的项目文件需要引用 NuGet 包 Microsoft.EntityFrameworkCore.Design。该工具只需要设计库，不需要从调用应用程序中引用，这就是为什么可以设置 PrivateAssets 特性的原因(项目文件 MigrationsConsoleApp/MigrationsConsoleApp.csproj)：

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
```

```
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.EntityFrameworkCore"
      Version="2.0.0" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer"
      Version="2.0.0" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Design"
      Version="2.0.0" PrivateAssets="All" />
  </ItemGroup>

  <ItemGroup>
    <ProjectReference Include="..\MigrationsLib\MigrationsLib.csproj" />
  </ItemGroup>

</Project>
```

### 26.11.4　创建迁移

有了这些，就可以创建一个初始迁移。使用以下命令，当前目录必须是库的目录——定义工具引用的目录。以下命令创建名为 InitMenus 的初始迁移。使用选项--startup-project 引用的启动项目包含初始代码，其中包括到服务器的连接字符串——或者使用从 ASP.NET Core Web 应用程序的默认项目模板中生成的 Main()方法，或实现 IDesignTimeContextFactory 的对象，如前一节所示：

```
> dotnet ef migrations add InitMenus --startup-project ../MigrationsConsoleApp
```

如果项目包含多个 DB 上下文，就需要提供附加选项--context，并提供 DB 上下文类的名称。

运行此命令，会创建一个带有快照的 Migrations 文件夹，以基于模型创建完整的数据库模式(代码文件 MigrationsLib/Migration/MenusContextModelSnapshot.cs)：

```
[DbContext(typeof(MenusContext))]
partial class MenusContextModelSnapshot : ModelSnapshot
{
  protected override void BuildModel(ModelBuilder modelBuilder)
  {
#pragma warning disable 612, 618
    modelBuilder
      .HasDefaultSchema("mc")
      .HasAnnotation("ProductVersion", "2.0.0-rtm-26452")
      .HasAnnotation("SqlServer:ValueGenerationStrategy",
        SqlServerValueGenerationStrategy.IdentityColumn);

    modelBuilder.Entity("MigrationsLib.Menu", b =>
    {
      b.Property<int>("MenuId")
        .ValueGeneratedOnAdd();

      b.Property<int>("MenuCardId");

      b.Property<decimal>("Price")
        .HasColumnType("Money");

      b.Property<string>("Text")
        .HasMaxLength(120);

      b.HasKey("MenuId");

      b.HasIndex("MenuCardId");

      b.ToTable("Menus");
    });

    modelBuilder.Entity("MigrationsLib.MenuCard", b =>
    {
      b.Property<int>("MenuCardId")
        .ValueGeneratedOnAdd();

      b.Property<bool>("IsDeleted");

      b.Property<DateTime>("LastUpdated");
```

```
          b.Property<string>("Title")
            .HasMaxLength(50);

          b.HasKey("MenuCardId");

          b.ToTable("MenuCards");
        });

        modelBuilder.Entity("MigrationsLib.Menu", b =>
        {
          b.HasOne("MigrationsLib.MenuCard", "MenuCard")
            .WithMany("Menus")
            .HasForeignKey("MenuCardId")
            .OnDelete(DeleteBehavior.Cascade);
        });
#pragma warning restore 612, 618
    }
}
```

对于每次迁移，都会创建一个从基类 Migration 派生的迁移类。这个基类定义了 Up 和 Down 方法，以允许将迁移应用到这个迁移版本中，或者向后一步(代码文件 MigrationsLib/<version>_InitialMenus.cs)：

```
public partial class InitialMenus : Migration
{
  protected override void Up(MigrationBuilder migrationBuilder)
  {
    migrationBuilder.EnsureSchema(name: "mc");

    migrationBuilder.CreateTable(
      name: "MenuCards",
      schema: "mc",
      columns: table => new
      {
        MenuCardId = table.Column<int>(type: "int", nullable: false)
          .Annotation("SqlServer:ValueGenerationStrategy",
            SqlServerValueGenerationStrategy.IdentityColumn),
       IsDeleted = table.Column<bool>(type: "bit", nullable: false),
       LastUpdated = table.Column<DateTime>(type: "datetime2",
         nullable: false),
       Title = table.Column<string>(type: "nvarchar(50)", maxLength: 50,
         nullable: true)
      },
      constraints: table =>
      {
        table.PrimaryKey("PK_MenuCards", x => x.MenuCardId);
      });

    migrationBuilder.CreateTable(
      name: "Menus",
      schema: "mc",
      columns: table => new
      {
        MenuId = table.Column<int>(type: "int", nullable: false)
          .Annotation("SqlServer:ValueGenerationStrategy",
            SqlServerValueGenerationStrategy.IdentityColumn),
        MenuCardId = table.Column<int>(type: "int", nullable: false),
        Price = table.Column<decimal>(type: "Money", nullable: false),
        Text = table.Column<string>(type: "nvarchar(120)", maxLength: 120,
          nullable: true)
      },
      constraints: table =>
      {
        table.PrimaryKey("PK_Menus", x => x.MenuId);
        table.ForeignKey(
          name: "FK_Menus_MenuCards_MenuCardId",
          column: x => x.MenuCardId,
          principalSchema: "mc",
          principalTable: "MenuCards",
          principalColumn: "MenuCardId",
          onDelete: ReferentialAction.Cascade);
      });

      migrationBuilder.CreateIndex(
        name: "IX_Menus_MenuCardId",
        schema: "mc",
        table: "Menus",
        column: "MenuCardId");
```

```
    }

    protected override void Down(MigrationBuilder migrationBuilder)
    {
      migrationBuilder.DropTable(
      name: "Menus",
      schema: "mc");

      migrationBuilder.DropTable(
      name: "MenuCards",
      schema: "mc");
    }
  }
```

在对模型进行更改之后，例如向 Menu 类添加 Allergens(代码文件 MigrationsLib/Menu.cs)：

```
public class Menu
{
  public int MenuId { get; set; }
  public string Text { get; set; }
  public decimal Price { get; set; }
  public string Allergens { get; set; }
  public int MenuCardId { get; set; }
  public MenuCard MenuCard { get; set; }
  public override string ToString() => Text;
}
```

就需要一个新迁移：

```
> dotnet ef migrations add AddAllergens --startup-project ..\MigrationsConsoleApp
```

使用新的迁移，将更新快照类，以显示当前状态，并添加新的 Migration 类型，以使用 Up 和 Down 方法添加和删除 allergen 列(代码文件 MigrationsLib/<version>_AddAllergens.cs)：

```
public partial class AddAllergens : Migration
{
  protected override void Up(MigrationBuilder migrationBuilder)
  {
    migrationBuilder.AddColumn<string>(
      name: "Allergens",
      schema: "mc",
      table: "Menus",
      type: "nvarchar(max)",
      nullable: true);
  }

  protected override void Down(MigrationBuilder migrationBuilder)
  {
    migrationBuilder.DropColumn(
      name: "Allergens",
      schema: "mc",
      table: "Menus");
  }
}
```

在应用新的迁移之前，请注意构建库。否则，新的迁移就可能是空的。

**注意：**
对于所做的每一个更改，都可以创建另一个迁移。新的迁移只定义了从旧版本到新版本所需的更改。如果客户的数据库需要从任何早期版本更新，那么在迁移数据库时将调用必要的迁移。

在开发过程中，可能会出现许多生产中不需要的迁移。只需要为可能在客户站点上运行的所有版本保留迁移。要从开发时删除迁移，可以调用 dotnet ef migrations remove 来删除最新的迁移代码。然后添加新的较大迁移，其中包含自上次迁移以来的所有更改。

### 26.11.5　以编程方式应用迁移

配置好迁移后，可以直接在应用程序中启动数据库的迁移过程。为此，控制台应用程序配置为使用依赖注入容器来检索 DB 上下文，然后调用 Database 属性的 Migrate 方法(代码文件 MigrationsConsoleApp/Program.cs)：

```
class Program
```

```
{
  private const string ConnectionString =
    @"server=(localdb)\mssqllocaldb;database=ProCSharpMigrations;" +
    @"trusted_connection=true";

  static void Main(string[] args)
  {
    RegisterServices();
    var context = Container.GetService<MenusContext>();
    context.Database.Migrate();
  }

  private static void RegisterServices()
  {
    var services = new ServiceCollection();
    services.AddDbContext<MenusContext>(options =>
      options.UseSqlServer(ConnectionString));
    Container = services.BuildServiceProvider();
  }
  public static IServiceProvider Container { get; private set; }
}
```

如果数据库还不存在，那么 Migrate 方法将创建数据库——使用模型定义的模式——以及一个__EFMigrationsHistory 表，该表列出了已应用到数据库的所有迁移。不能像前面那样使用 EnsureCreated 方法来创建数据库，因为该方法不向数据库应用迁移信息。

使用现有的数据库，数据库将更新到迁移的当前版本。通过编程，可以使用 GetMigrations 方法在应用程序中获得所有可用的迁移。要查看所有应用的迁移，可以使用 GetAppliedMigrations 方法。对于数据库中丢失的所有迁移，请使用 GetPendingMigrations 方法。

### 26.11.6 应用迁移的其他方法

除了以编程方式应用迁移之外，还可以使用命令行来应用迁移：

```
> dotnet ef database update --startup-project ../MigrationsConsoleApp
```

该命令将最新的迁移应用到数据库。还可以向该命令提供迁移的名称，以便将数据库放到迁移的特定版本中。

如果数据库管理员需要对数据库进行完全控制，且不允许进行编程更改，不允许对.NET Core CLI 命令行之类的工具进行任何更改，就可以创建一个 SQL 脚本，并将其提交或自己使用。

下面的命令行创建 SQL 脚本 migrationsscript.sql，其中包括从最初创建的数据库到最近的迁移。还可以为脚本中应该应用的迁移范围提供特定的 from/to 值：

```
> dotnet ef migrations script --output migrationsscript.sql
--startup-project ..\MigrationsConsoleApp
```

## 26.12 小结

本章介绍了 EF Core 的特性，学习了对象上下文如何了解检索和更新的实体，以及变更如何写入数据库。还讨论了迁移如何在 C#代码中用于创建和更改数据库模式。至于模式的定义，本章论述了如何使用数据注释进行数据库映射，流利 API 提供了比注释更多的功能。

本章阐述了多个用户处理同一记录时应对冲突的可能性，以及隐式或显式地使用事务，进行更多的事务控制。

本章论述了 EF Core 2.0 的新功能，例如已编译的查询、全局查询过滤器、表的分割和拥有的实体，用户可以开始使用它们了。

下一章介绍.NET 的全球化和本地化，包括使用区域化特定的日期、时间和数字格式，以及为不同语言定义不同文本的资源。

# 第 27 章

# 本　地　化

**本章要点**

- 数字和日期的格式化
- 为本地化内容使用资源
- 本地化 ASP.NET Core Web 应用程序
- 本地化通用 Windows 应用程序

**本章源代码下载地址(wrox.com)：**

打开 www.wrox.com 的 Download Code 选项卡可下载本章源代码。源代码也可以在 Localization 目录的 https://github.com/ProfessionalCSharp/ProfessionalCSharp7 中找到。本章代码分为以下几个主要的示例文件：

- NumberAndDateFormatting
- SortingDemo
- CreateResource
- UWPCultureDemo
- ResourcesDemo
- WebApplicationSample
- ASPNETCoreMVCSample
- UWPLocalization

## 27.1　全球市场

价值 1.25 亿美元的 NASA 的火星气象卫星在 1999 年 9 月 23 日失踪了，其原因是一个工程组为一个关键的太空操作使用了米制单位，而另一个工程组以英寸为单位。当编写的应用程序要在世界各国发布时，必须考虑不同的区域性和区域。

不同的区域性在日历、数字和日期格式上各不相同。按照字母 A～Z 给字符串排序也会导致不同的结果，因为存在不同的文化差异。为了使应用程序可应用于全球市场，就必须对应用程序进行全球化和本地化。

本章将介绍.NET 应用程序的全球化和本地化。全球化(globalization)用于国际化的应用程序：使应用程序可

以在国际市场上销售。采用全球化策略，应用程序应根据区域性、不同的日历等支持不同的数字和日期格式。本地化(localization)用于为特定的区域性翻译应用程序。而字符串的翻译可以使用资源，如.NET 资源或 WPF 资源字典。

.NET 支持 Windows 和 Web 应用程序的全球化和本地化。要使应用程序全球化，可以使用 System.Globalization 名称空间中的类；要使应用程序本地化，可以使用 System.Resources 名称空间支持的资源。

# 27.2 System.Globalization 名称空间

System.Globalization 名称空间包含了所有的区域性和区域类，以支持不同的日期格式、不同的数字格式，甚至由 GregorianCalendar 类、HebrewCalendar 类和 JapaneseCalendar 类等表示的不同日历。使用这些类可以根据不同的地区显示不同的表示法。

本节讨论使用 System.Globalization 名称空间时要考虑的如下问题：

- Unicode 问题
- 区域性和区域
- 显示所有区域性及其特征的例子
- 排序

## 27.2.1 Unicode 问题

因为一个 Unicode 字符有 16 位，所以共有 65 536 个 Unicode 字符。这对于当前在信息技术中使用的所有语言够用吗？例如，汉语就需要 80 000 多个字符。但是，Unicode 可以解决这个问题。使用 Unicode 必须区分基本字符和组合字符。可以给一个基本字符添加若干个组合字符，组成一个可显示的字符或一个文本元素。

例如，冰岛的字符 Ogonek 可以使用基本字符 0x006F(拉丁小写字母 o)、组合字符 0x0328(组合 Ogonek)和 0x0304(组合 Macron)组合而成，如图 27-1 所示。组合字符定义的范围是 0x0300~0x0345，对于美国和欧洲市场，预定义字符有助于处理特殊的字符。字符 Ogonek 也可以用预定义字符 0x01ED 来定义。

0x01ED = 0x006F + 0x0328 + 0x0304

图 27-1

对于亚洲市场，只有汉语需要 80 000 多个字符，但没有这么多的预定义字符。在亚洲语言中，总是要处理组合字符。其问题在于获取显示字符或文本元素的正确数字，得到基本字符而不是组合字符。System.Globalization 名称空间提供的 StringInfo 类可以用于处理这个问题。

表 27-1 列出了 StringInfo 类的静态方法，这些方法有助于处理组合字符。

表 27-1

| 方 法 | 说 明 |
| --- | --- |
| GetNextTextElement( ) | 返回指定字符串的第一个文本元素(基本字符和所有的组合字符) |
| GetTextElementEnumerator( ) | 返回一个允许迭代字符串中所有文本元素的 TextElementEnumerator 对象 |
| ParseCombiningCharacters( ) | 返回一个引用字符串中所有基本字符的整型数组 |

**注意：**

一个显示字符可以包含多个 Unicode 字符。要解决这个问题，如果编写的应用程序要在国际市场销售，就不应使用数据类型 char，而应使用 string。string 可以包含由基本字符和组合字符组成的文本元素，而 char 不具备该作用。

### 27.2.2 区域性和区域

世界分为多个区域性和区域，应用程序必须知道这些区域性和区域的差异。区域性是基于用户的语言和文化习惯的一组首选项。RFC 4646(www.ietf.org/rfc/rfc4646.txt)定义了区域性的名称，这些名称根据语言和国家或区域的不同在世界各地使用。例如，en-AU、en-CA、en-GB 和 en-US 分别用于表示澳大利亚、加拿大、英国和美国的英语。

在 System.Globalization 名称空间中，最重要的类是 CultureInfo。这个类表示区域性，定义了日历、数字和日期的格式，以及和区域性一起使用的排序字符串。

RegionInfo 类表示区域设置(如货币)，说明该区域是否使用米制系统。在某些区域中，可以使用多种语言。例如，西班牙区域就有 Basque(eu-ES)、Catalan(ca-ES)、Spanish(es-ES)和 Galician(gl-ES)区域性。一个区域可以有多种语言，同样，一种语言也可以在多个区域使用；例如，墨西哥、西班牙、危地马拉、阿根廷和秘鲁等都使用西班牙语。

本章的后面将介绍一个示例应用程序，以说明区域性和区域的这些特征。

#### 1. 特定、中立和不变的区域性

在.NET Framework 中使用区域性，必须区分 3 种类型：特定、中立和不变的区域性。特定的区域性与真正存在的区域性相关，这种区域性用 RFC 4646 定义。特定的区域性可以映射到中立的区域性。例如，de 是特定区域性 de-AT、de-DE、de-CH 等的中立区域性。de 是德语(German)的简写，AT、DE 和 CH 分别是奥地利(Austria)、德国(Germany)和瑞士(Switzerland)等国家的简写。

在翻译应用程序时，通常不需要为每个区域进行翻译，因为奥地利和瑞士等国使用的德语没有太大的区别。所以可以使用中立的区域性来本地化应用程序，而不需要使用特定的区域性。

不变的区域性独立于真正的区域性。在文件中存储格式化的数字或日期，或通过网络把它们发送到服务器上时，最好使用独立于任何用户设置的区域性。

图 27-2 显示了区域性类型的相互关系。

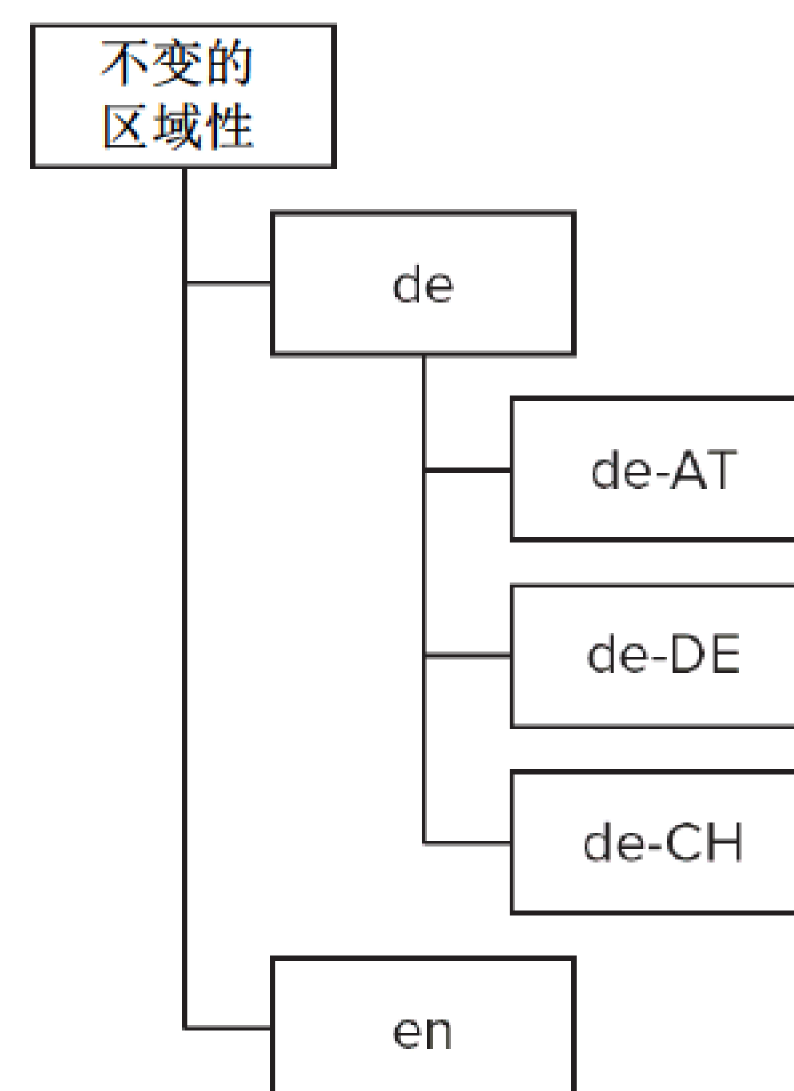


图 27-2

#### 2. CurrentCulture 和 CurrentUICulture

设置区域性时，必须区分用户界面的区域性和数字及日期格式的区域性。区域性与线程相关，通过这两种区域性类型，就可以把两种区域性设置应用于线程。CultureInfo 类提供了静态属性 CurrentCulture 和 CurrentUICulture。CurrentCulture 属性用于设置与格式化和排序选项一起使用的区域性，而 CurrentUICulture 属性用于设置用户界面的语言。

使用 Windows 设置中的 Time & Language，再从中选择 REGION & LANGUAGE 选项，用户就可以在 Windows 10 操作系统中安装其他语言，如图 27-3 所示。配置为默认的语言是当前的 UI 区域性。

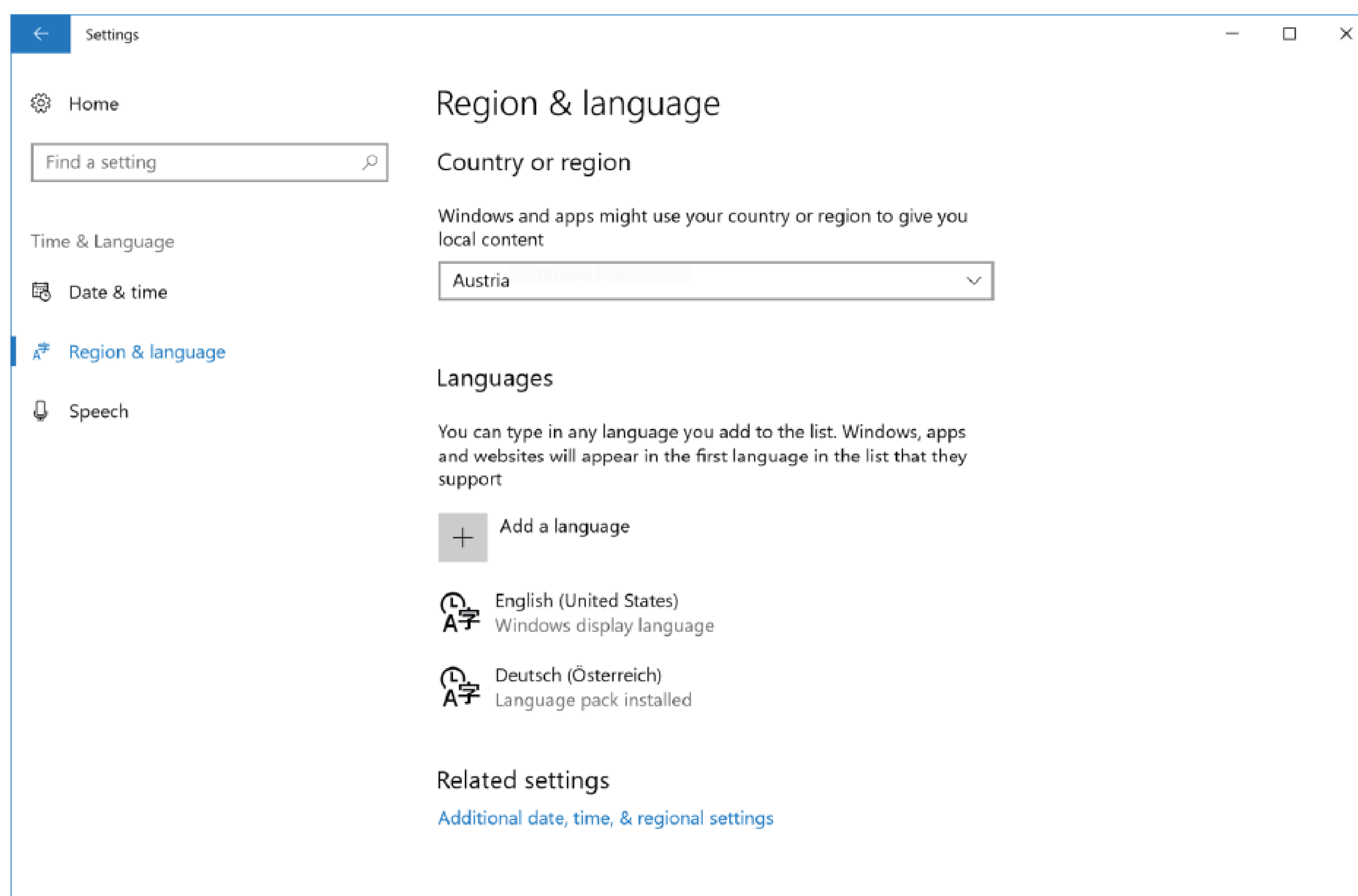



图 27-3

要改变当前的区域性，可以使用对话框中的 Additional date, time, & regional settings 链接，如图 27-3 所示。其中，单击 Change Date, Time, or Number Formats 选项，查看如图 27-4 所示的对话框。格式的语言设置会影响当前的区域性。也可以改变独立于区域性的数字格式、时间格式、日期格式的默认设置。

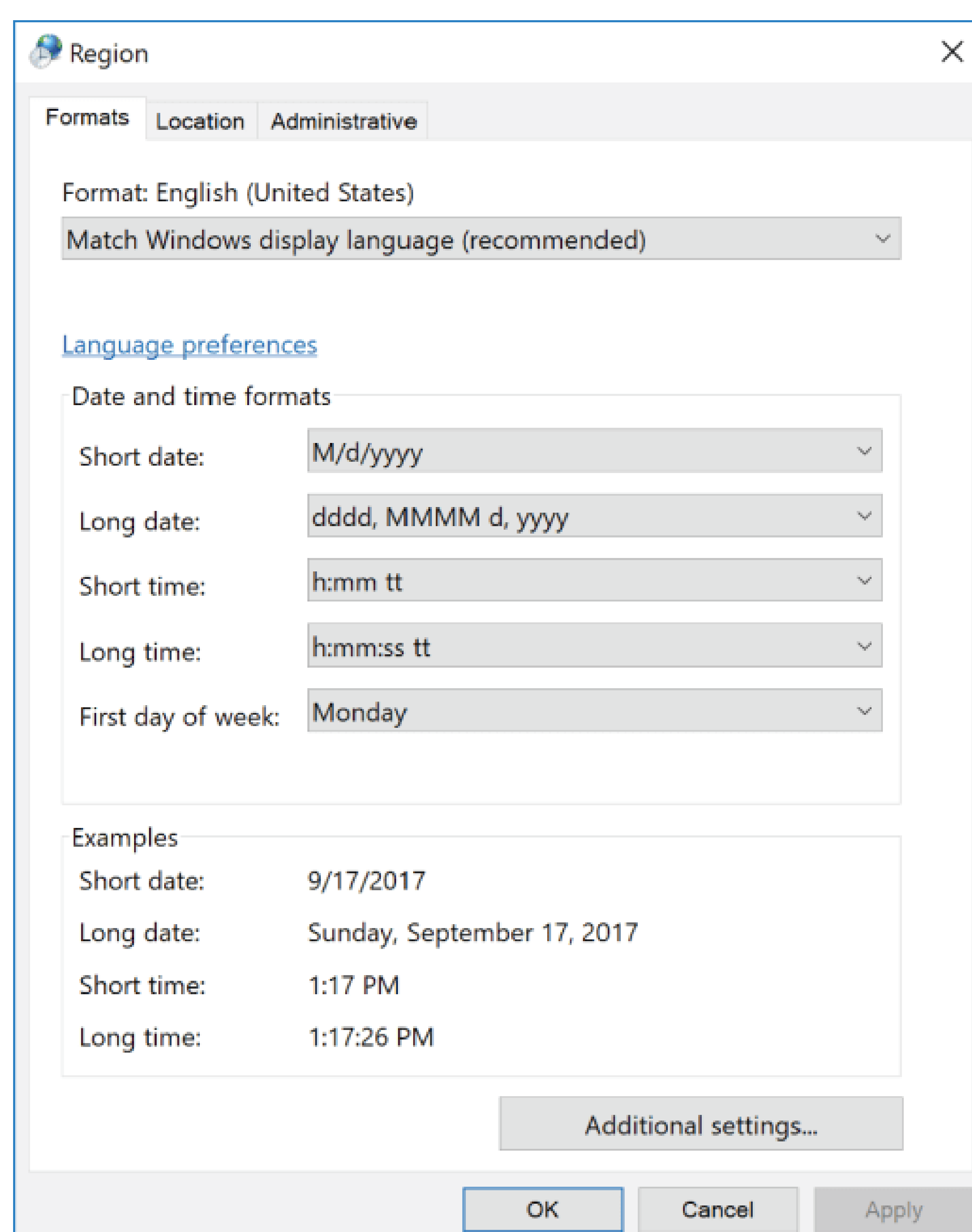



图 27-4

这些设置都提供了很好的默认值，在许多情况下，不需要改变默认行为。如果需要改变区域性，只需要把

线程的两个区域性改为 Spanish 区域性，如下面的代码片段所示(使用名称空间 System.Globalization)：

```
var ci = new CultureInfo("es-ES");
CultureInfo.CurrentCulture = ci;
CultureInfo.CurrentUICulture = ci;
```

前面已学习了区域性的设置，下面讨论 CurrentCulture 设置对数字和日期格式化的影响。

### 3. 数字格式化

System 名称空间中的数字结构 Int16、Int32 和 Int64 等都有一个重载的 ToString()方法。这个方法可以根据区域设置创建不同的数字表示法。对于 Int32 结构，ToString()方法有下述 4 个重载版本：

```
public string ToString();
public string ToString(IFormatProvider);
public string ToString(string);
public string ToString(string, IFormatProvider);
```

不带参数的 ToString()方法返回一个没有格式化选项的字符串，也可以给 ToString()方法传递一个字符串和一个实现 IFormatProvider 接口的类。

该字符串指定表示法的格式，而这个格式可以是标准数字格式化字符串或者图形数字格式化字符串。对于标准数字格式化，字符串是预定义的，其中 C 表示货币符号，D 表示输出为小数，E 表示输出用科学计数法表示，F 表示定点输出，G 表示一般输出，N 表示输出为数字，X 表示输出为十六进制数。对于图形数字格式化字符串，可以指定位数、节和组分隔符、百分号等。图形数字格式字符串###，###表示：两个三位数块被一个组分隔符分开。

IFormatProvider 接口由 NumberFormatInfo、DateTimeFormatInfo 和 CultureInfo 类实现。这个接口定义了 GetFormat()方法，它返回一个格式对象。

NumberFormatInfo 类可以为数字定义自定义格式。使用 NumberFormatInfo 类的默认构造函数，可以创建独立于区域性的对象或不变的对象。使用这个类的属性，可以改变所有格式化选项，如正号、百分号、数字组分隔符和货币符号等。从静态属性 InvariantInfo 返回一个与区域性无关的只读 NumberFormatInfo 对象。NumberFormatInfo 对象的格式化值取决于当前线程的 CultureInfo 类，该线程从静态属性 CurrentInfo 返回。

示例代码 NumberAndDateFormatting 使用如下名称空间：

```
System
System.Globalization
```

下一个示例使用一个简单的控制台应用程序((.NET Core)项目。在这段代码中，第一个示例显示了在当前线程的区域性格式(这里是 English-US，是操作系统的设置)中所显示的数字。第二个示例使用了带有 IFormatProvider 参数的 ToString()方法。CultureInfo 类实现 IFormatProvider 接口，所以创建一个使用法国区域性的 CultureInfo 对象。第 3 个示例改变了当前线程的区域性。使用 CultureInfo 实例的 CurrentCulture 属性，把区域性改为德国(代码文件 NumberAndDateFormatting/Program.cs)：

```
public static void NumberFormatDemo()
{
  int val = 1234567890;
  // culture of the current thread
  Console.WriteLine(val.ToString("N"));
  // use IFormatProvider
  Console.WriteLine(val.ToString("N", new CultureInfo("fr-FR")));
  // change the current culture
  CultureInfo.CurrentCulture = new CultureInfo("de-DE");
  Console.WriteLine(val.ToString("N"));
}
```

可以把这个结果与前面列举的美国、英国、法国和德国区域性的结果进行比较。

```
1,234,567,890.00
1 234 567 890,00
1.234.567.890,00
```

#### 4. 日期格式化

对于日期，Visual Studio 也提供了与数字相同的支持。DateTime 结构有一些把日期转换为字符串的 ToString 方法的重载。可以传送字符串格式并指定另一种区域性：

```
public string ToString();
public string ToString(IFormatProvider);
public string ToString(string);
public string ToString(string, IFormatProvider);
```

使用 ToString()方法的字符串参数，可以指定预定义格式字符或自定义格式字符串，把日期转换为字符串。DateTimeFormatInfo 类指定了可能的值。DateTimeFormatInfo 类指定的格式字符串有不同的含义。例如，D 表示长日期格式，d 表示短日期格式，ddd 表示一星期中某一天的缩写，dddd 表示一星期中某一天的全称，yyyy 表示年份，T 表示长时间格式，t 表示短时间格式。使用 IFormatProvider 参数可以指定区域性。使用不带 IFormatProvider 参数的重载方法，表示所使用的是当前线程的区域性(代码文件 NumberAndDateFormatting/Program.cs)：

```
public static void DateFormatDemo()
{
  var d = new DateTime(2017, 09, 17);
  // current culture
  Console.WriteLine(d.ToLongDateString());
  // use IFormatProvider
  Console.WriteLine(d.ToString("D", new CultureInfo("fr-FR")));
  // use current culture
  Console.WriteLine($"{CultureInfo.CurrentCulture}: {d:D}");
  CultureInfo.CurrentCulture = new CultureInfo("es-ES");
  Console.WriteLine($"{CultureInfo.CurrentCulture}: {d:D}");
}
```

这个示例程序的结果说明了使用线程的当前区域性的 ToLongDateString()方法，其中给 ToString()方法传递一个 CultureInfo 实例，则显示其法国版本；把线程的 CurrentCulture 属性改为 es-ES，则显示其西班牙版本，如下所示。

```
Sunday, September 17, 2017
dimanche 17 septembre 2017
en-US: Sunday, September 17, 2017
es-ES: domingo, 17 de septiembre de 2017
```

### 27.2.3 使用区域性

为了全面介绍区域性，下面使用一个 UWP 应用程序示例，该应用程序列出所有的区域性，描述区域性属性的不同特征。在 UI 的左边，树视图用于显示所有的区域性。右边是一个用户控件，显示了所选区域性和区域的相关信息。图 27-5 显示了该应用程序在 Visual Studio 2017 Designer 中的用户界面。

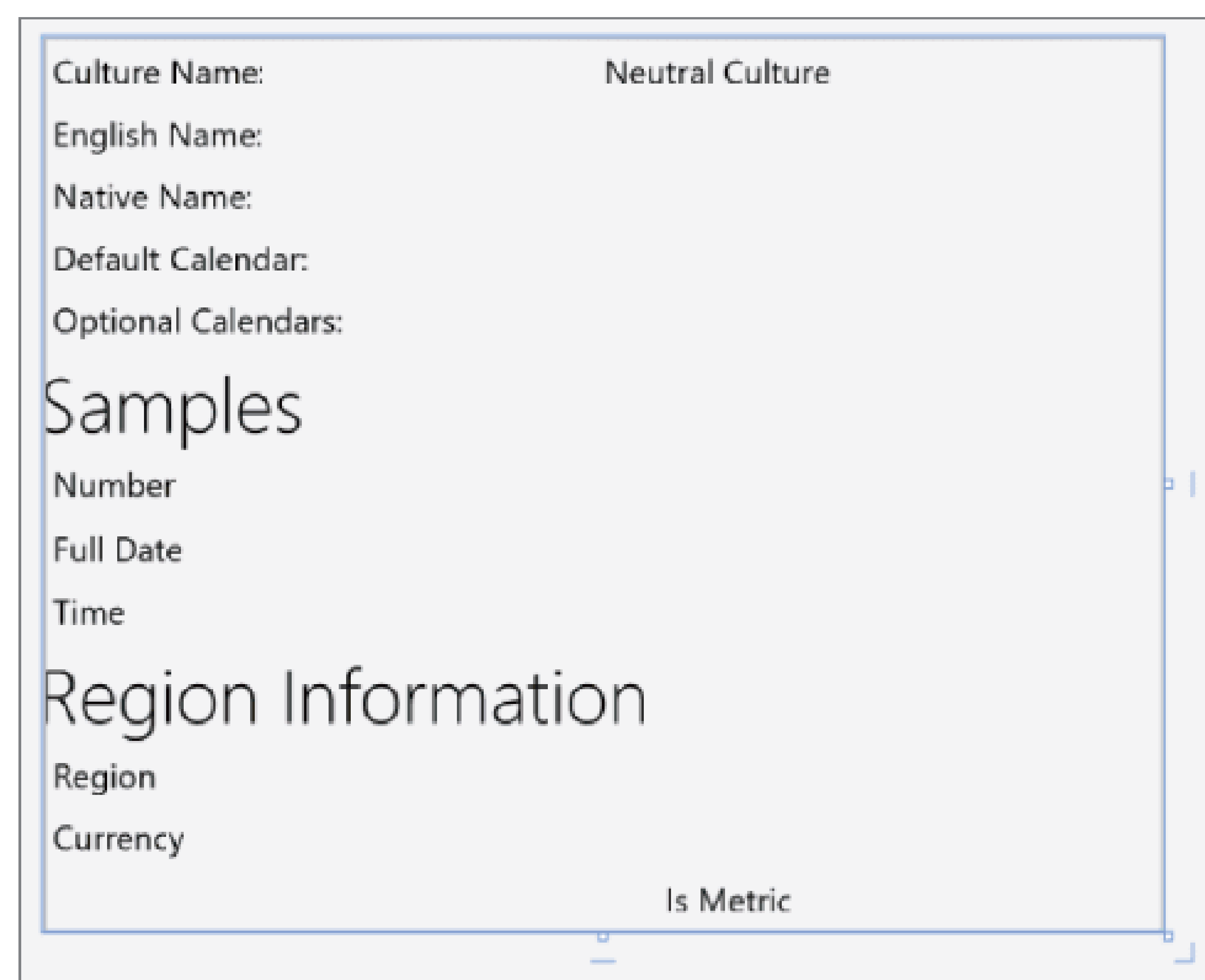



图 27-5

**注意：**

示例代码使用了树控件。Windows 10 运行库不包括 Fall Creators Update 2017 中的树视图控件，但预计以后会有树视图控件。有几个第三方的树控件可以用于 UWP。微软的 Windows Universal Samples 在 https://github.com/Microsoft/Windows-universal-samples 中，它也包含树视图控件。此控件用于这个 UWP 示例。因为这个控件是用 C++代码编写的，所以需要使用 Universal Windows Platform 开发工作负载安装 C++组件。

在应用程序的初始化阶段，所有可用的区域性都添加到应用程序左边的 TreeView 控件中。这个初始化工作在 SetupCultures()方法中进行，该方法在 CulturesViewModel 类的构造函数中调用(代码文件 UWPCultureDemo/CulturesViewModel.cs)：

```
public CulturesViewModel()
{
  SetupCultures();
}
```

对于在用户界面上显示的数据，创建自定义类 CultureData。这个类可以绑定到 TreeView 控件上，因为它的 SubCultures 属性包含一列 CultureData。因此 TreeView 控件可以遍历这个树状结构。CultureData 不包含子区域性，而包含数字、日期和时间的 CultureInfo 类型以及示例值。数字以适用于特定区域性的数字格式返回一个字符串，日期和时间也以特定区域性的格式返回字符串。CultureData 包含一个 RegionInfo 类来显示区域。对于一些中立区域性(例如 English)，创建 RegionInfo 会抛出一个异常，因为某些区域有特定的区域性。但是，对于其他中立区域性(例如 German)，可以成功创建 RegionInfo，并映射到默认的区域上。这里抛出的异常应这样处理(代码文件 UWPCultureDemo/CultureData.cs)：

```
public class CultureData
{
  public CultureInfo CultureInfo { get; set; }
  public List<CultureData> SubCultures { get; set; }
  double numberSample = 9876543.21;
  public string NumberSample => numberSample.ToString("N", CultureInfo);
  public string DateSample => DateTime.Today.ToString("D", CultureInfo);
  public string TimeSample => DateTime.Now.ToString("T", CultureInfo);

  public RegionInfo RegionInfo
  {
    get
    {
      RegionInfo ri;
      try
      {
        ri = new RegionInfo(CultureInfo.Name);
      }
      catch (ArgumentException)
      {
        // with some neutral cultures regions are not available
        return null;
      }
      return ri;
    }
  }
}
```

在 SetupCultures()方法中，通过静态方法 CultureInfo.GetCultures()获取所有区域性。给这个方法传递 CultureTypes.AllCultures，就会返回所有可用区域性的未排序数组。该数组按区域性名称来排序。有了排好序的区域性，就创建一个 CultureData 对象的集合，并分配 CultureInfo 和 SubCultures 属性。之后，创建一个字典，以快速访问区域性名称。

对于 UI 中应该显示的数据，创建一个 CultureData 对象列表，在执行完 foreach 语句后，该列表将包含树状视图中的所有根区域性。可以验证根区域性，以确定它们是否把不变的区域性作为其父区域性。不变的区域性把 LCID(Locale Identifier)设置为 127，每个区域性都有自己的唯一标识符，可用于快速验证。在代码段中，根区域性在 if 语句块中添加到 rootCultures 集合中。如果一个区域性把不变的区域性作为其父区域性，它就是根区域性。

如果区域性没有父区域性，它就会添加到树的根节点上。要查找父区域性，必须把所有区域性保存到一个字典中。相关内容参见前面章节，其中第 10 章介绍了字典，第 8 章介绍了 lambda 表达式。如果所迭代的区域性不是根区域性，它就添加到父区域性的 SubCultures 集合中。使用字典可以快速找到父区域性。在最后一步中，把根区域性赋予窗口的 DataContext，使根区域性可用于 UI(代码文件 UWPCultureDemo/CulturesViewModel.cs)：

```
private void SetupCultures()
{
  var cultureDataDict = CultureInfo.GetCultures(CultureTypes.AllCultures)
    .OrderBy(c => c.Name)
    .Select(c => new CultureData
    {
      CultureInfo = c,
      SubCultures = new List<CultureData>()
    })
    .ToDictionary(c => c.CultureInfo.Name);

  var rootCultures = new List<CultureData>();
  foreach (var cd in cultureDataDict.Values)
  {
    if (cd.CultureInfo.Parent.LCID == 127)
    {
      rootCultures.Add(cd);
    }
    else
    {
      if (cultureDataDict.TryGetValue(cd.CultureInfo.Parent.Name,
        out CultureData parentCultureData))
      {
        parentCultureData.SubCultures.Add(cd);
      }
      else
      {
        throw new InvalidOperationException(
          "parent culture not found");
      }
    }
  }

  foreach (var rootCulture in rootCultures.OrderBy(
    cd => cd.CultureInfo.EnglishName))
  {
    RootCultures.Add(rootCulture);
  }
}

public IList<CultureData> RootCultures { get; } = new List<CultureData>();
```

现在看看显示内容的 XAML 代码。一个树型视图用于显示所有的区域性。对于在树型视图内部显示的项，使用项模板。这个模板使用一个文本块，该文本框绑定到 CultureInfo 类的 EnglishName 属性上。(代码文件 UWPCultureDemo/MainPage.xaml)：

```
<tvc:TreeView x:Name="treeView1"
  IsMultiSelectCheckBoxEnabled="False"
  IsItemClickEnabled="True"
  SelectionChanged="{x:Bind OnSelectionChanged, Mode=OneTime}">
  <tvc:TreeView.ItemTemplate>
    <DataTemplate>
      <StackPanel Orientation="Horizontal" Height="40"
        Margin="{Binding Depth,
          Converter={StaticResource IntegerToIndConverter}}">
        <FontIcon x:Name="expandCollapseChevron"
          Glyph="{Binding IsExpanded,
            Converter={StaticResource ExpandCollapseGlyphConverter}}"
          Visibility="{Binding Data.SubCultures,
            Converter={StaticResource EmptyConverter}}"
          FontSize="12"
          Margin="12,8,12,8"
          FontFamily="Segoe MDL2 Assets" />
        <TextBlock Text="{Binding Data.CultureInfo.EnglishName}" />
      </StackPanel>
    </DataTemplate>
  </tvc:TreeView.ItemTemplate>
  <tvc:TreeView.ItemContainerTransitions>
    <TransitionCollection>
```

```
        <ContentThemeTransition />
        <ReorderThemeTransition />
        <EntranceThemeTransition IsStaggeringEnabled="False" />
      </TransitionCollection>
    </tvc:TreeView.ItemContainerTransitions>
  </tvc:TreeView>
```

在隐藏代码文件中，通过访问视图模型中的CultureData对象来初始化TreeView。使用CultureData对象，为TreeView创建TreeNode对象。TreeNode类定义了一个Data属性，在这个属性中，分配CultureData对象。TreeNode的Add方法允许添加子对象。通过递归调用本地函数AddSubNodes来添加子对象(代码文件UWPCultureDemo/MainPage.xaml.cs)：

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
  void AddSubNodes(TreeNode parent)
  {
    if (parent.Data is CultureData cd && cd.SubCultures != null)
    {
      foreach (var culture in cd.SubCultures)
      {
        var node = new TreeNode
        {
          Data = culture,
          ParentNode = parent
        };
        parent.Add(node);

        foreach (var subCulture in culture.SubCultures)
        {
          AddSubNodes(node);
        }
      }
    }
  }

  base.OnNavigatedTo(e);
  var rootNodes = ViewModel.RootCultures.Select(cd => new TreeNode
  {
    Data = cd
  });

  foreach (var node in rootNodes)
  {
    treeView1.RootNode.Add(node);
    AddSubNodes(node);
  }
}
```

在用户选择树中的一个节点时，就会调用 TreeView 类的 SelectedItemChanged 事件的处理程序。在下面的代码段中，这个处理程序在 OnSelectionChanged()方法中实现。在这个实现代码中，通过实现，把关联 ViewModel 的 SelectedCulture 属性设置为所选的 CultureData 对象(代码文件 UWPCultureDemo/MainPage.xaml.cs)：

```
private void OnSelectionChanged(object sender, SelectionChangedEventArgs e)
{
  ViewModel.SelectedCulture =
    (treeView1.SelectedItems?.FirstOrDefault() as TreeNode)?.Data
      as CultureData;
}
```

为了显示所选项的值，使用了几个 TextBlock 控件，它们绑定到 CultureData 类的 CultureInfo 属性上，从而绑定到从 CultureInfo 返回的 CultureInfo 类型的属性上，例如 Name、IsNeutralCulture、EnglishName 和 NativeName 等。要把从 IsNeutralCulture 属性返回的布尔值转换为 Visiblility 枚举值，并显示日历名称，应使用转换器(XAML 文件 UWPCultureDemo/CultureDetailUC.xaml)：

```
<TextBlock Grid.Row="0" Grid.Column="0" Text="Culture Name:" />
<TextBlock Grid.Row="0" Grid.Column="1"
  Text="{x:Bind CultureData.CultureInfo.Name, Mode=OneWay}"
  Width="100" />
<TextBlock Grid.Row="0" Grid.Column="2" Text="Neutral Culture"
  Visibility="{x:Bind CultureData.CultureInfo.IsNeutralCulture, Mode=OneWay}" />

<TextBlock Grid.Row="1" Grid.Column="0" Text="English Name:" />
```

```
<TextBlock Grid.Row="1" Grid.Column="1" Grid.ColumnSpan="2"
  Text="{x:Bind CultureData.CultureInfo.EnglishName, Mode=OneWay}" />

<TextBlock Grid.Row="2" Grid.Column="0" Text="Native Name:" />
<TextBlock Grid.Row="2" Grid.Column="1" Grid.ColumnSpan="2"
  Text="{x:Bind CultureData.CultureInfo.NativeName}" />

<TextBlock Grid.Row="3" Grid.Column="0" Text="Default Calendar:" />
<TextBlock Grid.Row="3" Grid.Column="1" Grid.ColumnSpan="2"
  Text="{x:Bind CultureData.CultureInfo.Calendar, Mode=OneWay
  Converter={StaticResource calendarConverter}}" />

<TextBlock Grid.Row="4" Grid.Column="0" Text="Optional Calendars:" />
<ListBox Grid.Row="4" Grid.Column="1" Grid.ColumnSpan="2"
  ItemsSource="{x:Bind CultureData.CultureInfo.OptionalCalendars}">
  <ListBox.ItemTemplate>
    <DataTemplate>
      <TextBlock Text="{Binding
        Converter={StaticResource calendarConverter}}" />
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

要显示日历文本，可使用实现了 IValueConverter 的对象。下面是 CalendarTypeToCalendarInformationConverter 类中 Convert 方法的实现代码，该实现代码使用类名和日历类型名称，给日历返回一个有用的值(代码文件 UWPCultureDemo/Converters/CalendarTypeToCalendarInformationConverter.cs)：

```
public object Convert(object value, Type targetType, object parameter,
  string language)
{
  if (value is Calendar cal)
  {
    var calText = new StringBuilder(50);
    calText.Append(cal.ToString());
    calText.Remove(0, 21); // remove the namespace
    calText.Replace("Calendar", "");

    if (cal is GregorianCalendar gregCal)
    {
      calText.Append($" {gregCal.CalendarType}");
    }

    return calText.ToString();
  }
  else
  {
    return null;
  }
}
```

CultureData 类包含的属性可以为数字、日期和时间格式显示示例信息，这些属性用下面的 TextBlock 元素绑定(XAML 文件 UWPCultureDemo/CultureDetailUC.xaml)：

```
<TextBlock Grid.Row="0" Grid.Column="0" Text="Number" />
<TextBlock Grid.Row="0" Grid.Column="1"
  Text="{x:Bind CultureData.NumberSample, Mode=OneWay}" />
<TextBlock Grid.Row="1" Grid.Column="0" Text="Full Date" />
<TextBlock Grid.Row="1" Grid.Column="1"
  Text="{x:Bind CultureData.DateSample, Mode=OneWay}" />
<TextBlock Grid.Row="2" Grid.Column="0" Text="Time" />
<TextBlock Grid.Row="2" Grid.Column="1"
  Text="{x:Bind CultureData.TimeSample, Mode=OneWay}" />
```

区域的信息用 XAML 代码的最后一部分显示。如果 RegionInfo 不可用，就隐藏整个 GroupBox。TextBlock 元素绑定了 RegionInfo 类型的 DisplayName、CurrencySymbol、ISOCurrencySymbol 和 IsMetric 属性：

```
<Grid Grid.Row="6" Grid.Column="0" Grid.ColumnSpan="3"
  Visibility="{x:Bind CultureData.RegionInfo, Mode=OneWay,
  Converter={StaticResource NullConverter}}">
  <!-- ... -->
  <TextBlock Grid.Row="0" Grid.Column="0" Text="Region Information"
    Style="{StaticResource SubheaderTextBlockStyle}" />
  <TextBlock Grid.Row="0" Grid.Column="1" Grid.ColumnSpan="2"
    Text="{x:Bind CultureData.RegionInfo.DisplayName, Mode=OneWay}" />
```

```
  <TextBlock Grid.Row="1" Grid.Column="0" Text="Currency" />
  <TextBlock Grid.Row="1" Grid.Column="1"
    Text="{x:Bind CultureData.RegionInfo.CurrencySymbol, Mode=OneWay}" />

  <TextBlock Grid.Row="1" Grid.Column="2"
    Text="{x:Bind CultureData.RegionInfo.ISOCurrencySymbol, Mode=OneWay}" />

  <TextBlock Grid.Row="2" Grid.Column="1" Text="Is Metric"
    Visibility="{x:Bind CultureData.RegionInfo.IsMetric, Mode=OneWay}" />
</Grid>
```

启动应用程序，在树型视图中就会看到所有的区域性，选择一个区域性后，就会列出该区域性的特征，如图 27-6 所示。

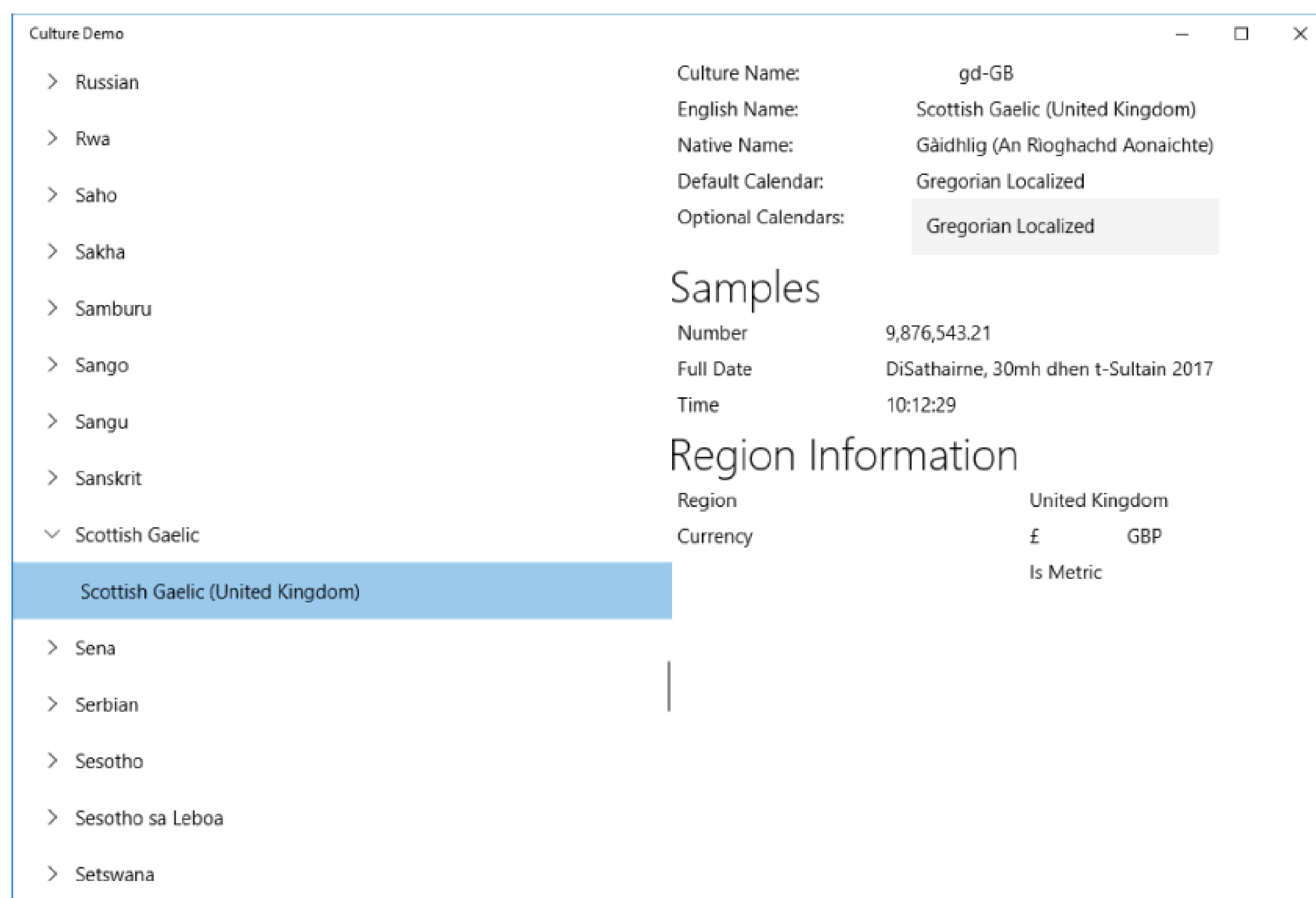



图 27-6

## 27.2.4　排序

SortingDemo 示例使用如下名称空间：

System

System.Collections

System.Collections.Generic

System.Globalization

排序字符串取决于区域性。在默认情况下，为排序而比较字符串的算法依赖于区域性。例如在芬兰，字符 V 和 W 就是相同的。为了说明芬兰的排序方式，下面的代码创建一个小型控制台应用程序示例，其中对数组中尚未排序的一些美国州名进行排序。

下面的 DisplayName()方法用于在控制台上显示数组或集合中的所有元素(代码文件 SortingDemo/Program.cs)：

```
public static void DisplayNames(string title, IEnumerable<string> names)
{
  Console.WriteLine(title);
  Console.WriteLine(string.Join("-", names));
  Console.WriteLine();
}
```

在 Main()方法中，在创建了包含一些美国州名的数组后，就把线程的 CurrentCulture 属性设置为 Finnish 区域性，这样，下面的 Array.Sort()方法就使用芬兰的排列顺序。调用 DisplayName()方法在控制台上显示所有

的州名：

```
public static void Main()
{
  string[] names = {"Alabama", "Texas", "Washington", "Virginia",
    "Wisconsin", "Wyoming", "Kentucky", "Missouri", "Utah",
    "Hawaii","Kansas", "Louisiana", "Alaska", "Arizona"};
  CultureInfo.CurrentCulture = new CultureInfo("fi-FI");
  Array.Sort(names);
  DisplayNames("Sorted using the Finnish culture", names);
  //...
}
```

在以芬兰排列顺序第一次显示美国州名后，数组将再次排序。如果希望排序独立于用户的区域性，就可以使用不变的区域性。在要将已排序的数组发送到服务器上或存储到某个地方时，就可以采用这种方式。

为此，给 Array.Sort()方法传递第二个参数。Sort()方法希望第二个参数是实现 IComparer 接口的一个对象。System.Collections 名称空间中的 Comparer 类实现 IComparer 接口。Comparer. DefaultInvariant 返回一个 Comparer 对象，该对象使用不变的区域性比较数组值，以进行独立于区域性的排序。

```
public static void Main()
{
  //...
  // sort using the invariant culture
  Array.Sort(names, System.Collections.Comparer.DefaultInvariant);
  DisplayNames("Sorted using the invariant culture", names);
}
```

这个程序的输出显示了用 Finnish 区域性进行排序的结果和独立于区域性的排序结果。在使用独立于区域性的排序方式时，Virginia 排在 Washington 的前面。用 Finnish 区域性进行排序时，Virginia 排在 Washington 的后面。

```
Sorted using the Finnish culture
Alabama-Alaska-Arizona-Hawaii-Kansas-Kentucky-Louisiana-Missouri-Texas-Utah-
Washington-Virginia-Wisconsin-Wyoming
Sorted using the invariant culture
Alabama-Alaska-Arizona-Hawaii-Kansas-Kentucky-Louisiana-Missouri-Texas-Utah-
Virginia-Washington-Wisconsin-Wyoming
```

**注意：**

如果对集合进行的排序应独立于区域性，该集合就必须用不变的区域性进行排序。在把排序结果发送给服务器或存储在文件中时，这种方式尤其有效。为了给用户显示排序的集合，最好用用户的区域性给它排序。

除了依赖区域设置的格式化和测量系统之外，文本和图片也可能因区域性的不同而有所变化。此时就需要使用资源。

# 27.3 资源

像图片或字符串表这样的资源可以放在资源文件或附属程序集中。在本地化应用程序时，这种资源非常有用，.NET 对本地化资源的搜索提供了内置支持。在说明如何使用资源本地化应用程序之前，先讨论如何创建和读取资源，而不需要考虑语言因素。

## 27.3.1 资源读取器和写入器

在.NET Core 中，资源读取器和写入器与完整的.NET 版本相比是有限的(在撰写本文时)。然而，在许多情形下(包括多平台支持)，资源读取器和写入器提供了必要的功能。

CreateResource 示例应用程序动态创建了一个资源文件，并从文件中读取资源。这个示例使用以下名称空间：

System
System.Collections

System.IO

System.Resources

ResourceWriter 允许创建二进制资源文件。写入器的构造函数需要一个使用 File 类创建的 Stream。利用 AddResource 方法添加资源(代码文件 CreateResource/Program.cs)：

```
private const string ResourceFile = "Demo.resources";
public static void CreateResource()
{
  FileStream stream = File.OpenWrite(ResourceFile);
  using (var writer = new ResourceWriter(stream))
  {
    writer.AddResource("Title", "Professional C#");
    writer.AddResource("Author", "Christian Nagel");
    writer.AddResource("Publisher", "Wrox Press");
  }
}
```

要读取二进制资源文件的资源，可以使用 ResourceReader。读取器的 GetEnumerator 方法返回一个 IDictionaryEnumerator，在以下 foreach 语句中使用它访问资源的键和值：

```
public static void ReadResource()
{
  FileStream stream = File.OpenRead(ResourceFile);
  using (var reader = new ResourceReader(stream))
  {
    foreach (DictionaryEntry resource in reader)
    {
      Console.WriteLine($"{resource.Key} {resource.Value}");
    }
  }
}
```

运行应用程序，返回写入二进制资源文件的键和值。如下一节所示，还可以使用命令行工具——资源文件生成器(resgen)——创建和转换资源文件。

## 27.3.2 使用资源文件生成器

资源文件包含图片、字符串表等条目。要创建资源文件，可以使用一般的文本文件，或者使用那些利用 XML 的.resX 文件。下面从一个简单的文本文件开始。

内嵌字符串表的资源可以使用一般的文本文件来创建。该文本文件只是把字符串赋予键。键是可以用来从程序中获取值的名称。键和值都可以包含空格。

这个例子显示了 Wrox.ProCSharp.Localization.MyResources.txt 文件中的一个简单字符串表：

```
Title = Professional C#
Chapter = Localization
Author = Christian Nagel
Publisher = Wrox Press
```

**注意：**

*在保存带 Unicode 字符的文本文件时，必须将文件和相应的编码一起保存。为此，可以在 Save As 对话框中选择 UTF8 编码。*

可以使用资源文件生成器(Resgen.exe)实用程序在 Wrox.ProCSharp.Localization.MyResources.txt 的外部创建一个资源文件，输入如下代码：

```
resgen Wrox.ProCSharp.Localization.MyResources.txt
```

这会创建 Wrox.ProCSharp.Localization.MyResources.resources 文件。得到的资源文件可以作为一个外部文件添加到程序集中，或者内嵌到 DLL 或 EXE 中。Resgen 还可以创建基于 XML 的.resX 资源文件。构建 XML 文件的一种简单方法是使用 Resgen 本身：

```
resgen Wrox.ProCSharp.Localization.MyResources.txt
Wrox.ProCSharp.Localization.MyResources.resX
```

这条命令创建了 XML 资源文件 Wrox.ProCSharp.Localization.MyResources.resX。Resgen 支持强类型化的资源。强类型化的资源用一个访问资源的类表示。这个类可以用 Resgen 实用程序的/str 选项创建：

```
resgen /str:C#,Wrox.ProCSharp.Localization,MyResources,MyResources.cs
Wrox.ProCSharp.Localization.MyResources.resX
```

在/str 选项中，按照语言、名称空间、类名和源代码文件名的顺序定义资源。

### 27.3.3 通过 ResourceManager 使用资源文件

默认情况下，资源文件都嵌入程序集。可以自定义它——例如，在项目文件中把带有 Remove 属性的 EmbeddedResource 元素添加到 ItemGroup 中，就可以从程序集中删除资源，如下所示：

```
<ItemGroup>
  <EmbeddedResource Remove="Resources\Messages.de.resx" />
</ItemGroup>
```

要了解资源文件如何使用 ResourceManager 类加载，创建一个控制台应用程序(.NET Core)，命名为 ResourcesDemo。这个示例使用以下名称空间：

System

System.Globalization

System.Reflection

System.Resources

创建一个 Resources 文件夹，在其中添加 Messages.resx 文件。Messages.resx 文件填充了 English-US 内容的键和值，例如键 GoodMorning 和值 Good Morning! 这是默认的语言。可以添加其他语言资源文件和命名约定，把区域性添加到资源文件中，例如，Messages.de.resx 表示德语，Messages.de-AT.resx 表示奥地利口音。

要访问嵌入式资源，使用 System.Resources 名称空间中的 ResourceManager 类和 NuGet 包 System.Resources.ResourceManager。实例化 ResourceManager 时，一个重载的构造函数需要资源的名称和程序集。应用程序的名称空间是 ResourcesDemo；资源文件在 Resources 文件夹中，它定义了子名称空间 Resources，其名称是 Messages.resx。它定义了名称 ResourcesDemo.Resources.Messages。可以使用 Program 类型的 GetTypeInfo 方法检索资源的程序集，它定义了一个 Assembly 属性。使用 resources 实例，GetString 方法返回从资源文件传递的键的值。给第二个参数传递一个区域性，例如 de-AT，就在 de-AT 资源文件中查找资源。如果没有找到，就提取中性语言 de，在 de 资源文件中查找资源。如果没有找到，就在没有指定区域性的默认资源文件中查找，返回值(代码文件 ResourcesDemo/Program.cs)：

```
var resources = new ResourceManager("ResourcesDemo.Resources.Messages",
  typeof(Program).GetTypeInfo().Assembly);
string goodMorning = resources.GetString("GoodMorning",
  new CultureInfo("de-AT"));
Console.WriteLine(goodMorning);
```

ResourceManager 构造函数的另一个重载版本只需要类的类型。这个 ResourceManager 查找 Program.resx 资源文件：

```
var programResources = new ResourceManager(typeof(Program));
Console.WriteLine(programResources.GetString("Resource1"));
```

### 27.3.4 System.Resources 名称空间

在举例之前，本节先复习一下 System.Resources 名称空间中处理资源的类：

- ResourceManager 类可以用于从程序集或资源文件中获取当前区域性的资源。使用 ResourceManager 类还可以获取特定区域性的 ResourceSet 类。
- ResourceSet 类表示特定区域性的资源。在创建 ResourceSet 类的实例时，它会枚举一个实现 IResourceReader 接口的类，并在散列表中存储所有的资源。
- IResourceReader 接口用于从 ResourceSet 中枚举资源。ResourceReader 类实现这个接口。

- ResourceWriter 类用于创建资源文件。ResourceWriter 类实现 IResourceWriter 接口。

# 27.4　使用 ASP.NET Core 本地化

> **注意：**
> 使用本地化与 ASP.NET Core，需要了解本章讨论的区域性和资源，以及如何创建 ASP.NET Core 应用程序。如果以前没有使用 ASP.NET Core 创建 ASP.NET Core Web 应用程序，就应该阅读第 30 章，再继续学习本章的这部分内容。

本地化 ASP.NET Core Web 应用程序时，可以使用 CultureInfo 类和资源，其方式类似于本章前面的内容，但有一些额外的问题需要解决。设置完整应用程序的区域性不能满足一般需求，因为用户来自不同的区域。所以有必要给到服务器的每个请求设置区域性。

如何知道用户的区域性呢？这有不同的选项。浏览器在每个请求的 HTTP 标题中发送首选语言。浏览器中的这个信息可以来自浏览器设置，或浏览器本身会检查安装的语言。另一个选项是定义 URL 参数，或为不同的语言使用不同的域名。可以在一些场景中使用不同的域名，例如为网站 www. cninnovation.com 使用英文版本，为 www.cninnovation.de 使用德语版本。但是 www.cninnovation.ch 呢?应该提供德语、法语和意大利语版本。这里，URL 参数，如 www.cninnovation.com/culture=de，会有所帮助。使用 www.cninnovation.com/de 的工作方式类似于定义特定路由的 URL 参数。另一个选择是允许用户选择语言，定义一个 cookie，来记住这个选项。

ASP.NET Core 支持所有这些场景。

## 27.4.1　注册本地化服务

为了开始注册操作，使用 Empty ASP.NET Core 项目模板创建一个新的 ASP.NET Core Web 应用程序。本项目利用以下依赖项和名称空间：

**依赖项**

Microsoft.AspNetCore.All

**名称空间**

Microsoft.AspNetCore
Microsoft.AspNetCore.Builder
Microsoft.AspNetCore.Hosting
Microsoft.AspNetCore.Http
Microsoft.AspNetCore.Localization
Microsoft.Extensions.DependencyInjection
Microsoft.Extensions.Localization
System
System.Globalization
System.Text.Encodings.Web

在 Startup 类中，需要调用 AddLocalization 扩展方法来注册本地化的服务(代码文件 WebApplicationSample/Startup.cs)：

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddLocalization(options => options.ResourcesPath =
    "CustomResources");
}
```

AddLocalization 方法为接口 IStringLocalizerFactory 和 IStringLocalizer 注册服务。在注册代码中，类型 ResourceManagerStringLocalizerFactory 注册为 singleton，StringLocalizer 注册短暂的生存期。类

ResourceManagerStringLocalizerFactory 是 ResourceManagerStringLocalizer 的一个工厂。这个类利用前面的 ResourceManager 类，从资源文件中检索字符串。

### 27.4.2 注入本地化服务

将本地化添加到服务集合后，就可以在 Startup 类的 Configure 方法中请求本地化。UseRequestLocalization 方法定义了一个重载版本，在其中可以传递 RequestLocalizationOptions。RequestLocalizationOptions 允许定制应该支持的区域性并设置默认的区域性。这里，DefaultRequestCulture 被设置为 en-us。类 RequestCulture 只是一个小包装，其中包含了用于格式化的区域性(它可以通过 Culture 属性来访问)和使用资源的区域性(UICulture 属性)。示例代码给 SupportedCultures 和 SupportedUICultures 接受 en-US、en.de-AT 和 de 区域性(代码文件 WebApplicationSample/Startup.cs)：

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env,
  IStringLocalizer<Startup> sr)
{
  //...
  var supportedCultures = new[]
  {
      new CultureInfo("en-US"),
      new CultureInfo("en"),
      new CultureInfo("de-AT"),
      new CultureInfo("de")
  };

  var options = new RequestLocalizationOptions
  {
    DefaultRequestCulture = new RequestCulture(new CultureInfo("en-US")),
    SupportedCultures = supportedCultures,
    SupportedUICultures = supportedCultures
  };
  app.UseRequestLocalization(options);
  //...
}
```

有了 RequestLocalizationOptions 设置，也设置属性 RequestCultureProviders。默认情况下配置 3 个提供程序：QueryStringRequestCultureProvider、CookieRequestCultureProvider 和 AcceptLanguage HeaderRequestCultureProvider。

### 27.4.3 区域性提供程序

下面详细讨论这些区域性提供程序。QueryStringRequestCultureProvider 使用查询字符串检索区域性。默认情况下，查询参数 culture 和 ui-culture 用于这个区域性提供程序，如下面的 URL 所示：

```
http://localhost:5000/?culture=de&ui-culture=en-US
http://localhost:5000/?culture=de&ui-culture=en-US
```

还可以通过设置 QueryStringRequestCultureProvider 的 QueryStringKey 和 UIQueryStringKey 属性来更改查询参数。

CookieRequestCultureProvider 定义了名为 ASPNET_CULTURE 的 cookie (可以使用 CookieName 属性设置)。检索这个 cookie 的值，来设置区域性。为了创建一个 cookie，并将其发送到客户端，可以使用静态方法 MakeCookieValue，从 RequestCulture 中创建一个 cookie，并将其发送到客户端。CookieRequestCultureProvider 使用静态方法 ParseCookieValue 获得 RequestCulture。

设置区域性的第三个选项是，可以使用浏览器发送的 HTTP 标题信息。发送的 HTTP 标题如下所示：

```
Accept-Language: en-us, de-at;q=0.8, it;q=0.7
```

AcceptLanguageHeaderRequestCultureProvider 使用这些信息来设置区域性。使用至多三个语言值，其顺序由 quality 值定义，找到与支持的区域性匹配的第一个值。

下面的代码片段现在使用请求的区域性生成HTML输出。首先，使用IRequestCultureFeature协定访问请求的区域性。实现接口IRequestCultureFeature的RequestCultureFeature使用匹配区域性设置的第一个区域性提供程序。如果URL定义了一个匹配区域性参数的查询字符串，就使用QueryStringRequestCultureProvider返回所请求的区

域性。如果URL不匹配，但收到名为ASPNET_CULTURE 的cookie，就使用CookieRequestCultureProvider，否则使用AcceptLanguageRequestCultureProvider。使用返回的RequestCulture的属性，把由此产生的、用户使用的区域性写入响应流。接着，使用当前的区域性把当前的日期写入流。这里使用的IStringLocalizer类型的变量需要一些更多的检查，如下所示(代码文件WebApplicationSample/Startup.cs)：

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env,
  IStringLocalizer<Startup> sr)
{
  //...
  app.Run(async context =>
  {
    IRequestCultureFeature requestCultureFeature =
      context.Features.Get<IRequestCultureFeature>();
    RequestCulture requestCulture = requestCultureFeature.RequestCulture;
    var today = DateTime.Today;
    await context.Response.WriteAsync("<h1>Sample Localization</h1>");
    await context.Response.WriteAsync(
      $"<div>{requestCulture.Culture} {requestCulture.UICulture}</div>");
    await context.Response.WriteAsync($"<div>{today:D}</div>");

    //...
  });
}
```

### 27.4.4 在 ASP.NET Core 中使用资源

资源文件可以用于 ASP.NET Core。样例项目添加了 CustomResources 文件夹并在其中添加了文件Startup.resx。资源的本地化版本用 Startup.de.resx 和 Startup.de-AT.resx 提供。

在注入本地化服务时，存储资源的文件夹名称用选项定义(代码文件 WebApplicationSample/Startup.cs)：

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddLocalization(
    options => options.ResourcesPath = "CustomResources");
}
```

在依赖注入中，IStringLocalizer <Startup>注入为 Configure 方法的一个参数。使用泛型类型的 Startup 参数，在 resources 目录中找到一个具有相同名称的资源文件，它匹配 Startup.resx。

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env,
  IStringLocalizer<Startup> sr)
{
  //...
}
```

**注意：**
依赖注入参见第 20 章。

下面的代码片段利用 IStringLocalizer<Startup>类型的变量 sr，通过一个索引器和 GetString 方法访问资源message1。资源 message2 使用字符串格式占位符，它用 GetString 方法的一个重载版本注入，其中可以传递任何数量的参数：

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env,
  IStringLocalizer<Startup> sr)
{
  //...
  app.Run(async context =>
  {
    //...
    await context.Response.WriteAsync(
      $"<div>{HtmlEncoder.Default.Encode(sr["message1"])}</div>");
    await context.Response.WriteAsync(
      $"<div>{HtmlEncoder.Default.Encode(sr.GetString("message1"))}</div>");
    await context.Response.WriteAsync(
      $"<div>{HtmlEncoder.Default.Encode(sr.GetString("message2",
        requestCulture.Culture, requestCulture.UICulture))}</div>");
  });
```

```
}
```

**注意：**

在示例代码中，HtmlEncoder 用于修改资源的输出，之后把它传递给 HttpResponse 的 WriteAsync 方法。通过这个编码器，资源值会转换为 HTML 格式，正确显示特殊的字符，例如德语中的 ü 和 ß。进行 HTML 编码后，这些字符会转换为 HTML 格式ü 和ß。

message1 的资源是一个简单的字符串；message2 的资源用字符串格式占位符定义：

```
Using culture {0} and UI culture {1}
```

**注意：**

在资源中使用格式化的字符串，就不能使用带插值字符串的新语法。占位符中与插值字符串一起使用的变量或表达式不能用于资源。

运行 Web 应用程序，得到的视图如图 27-7 所示。

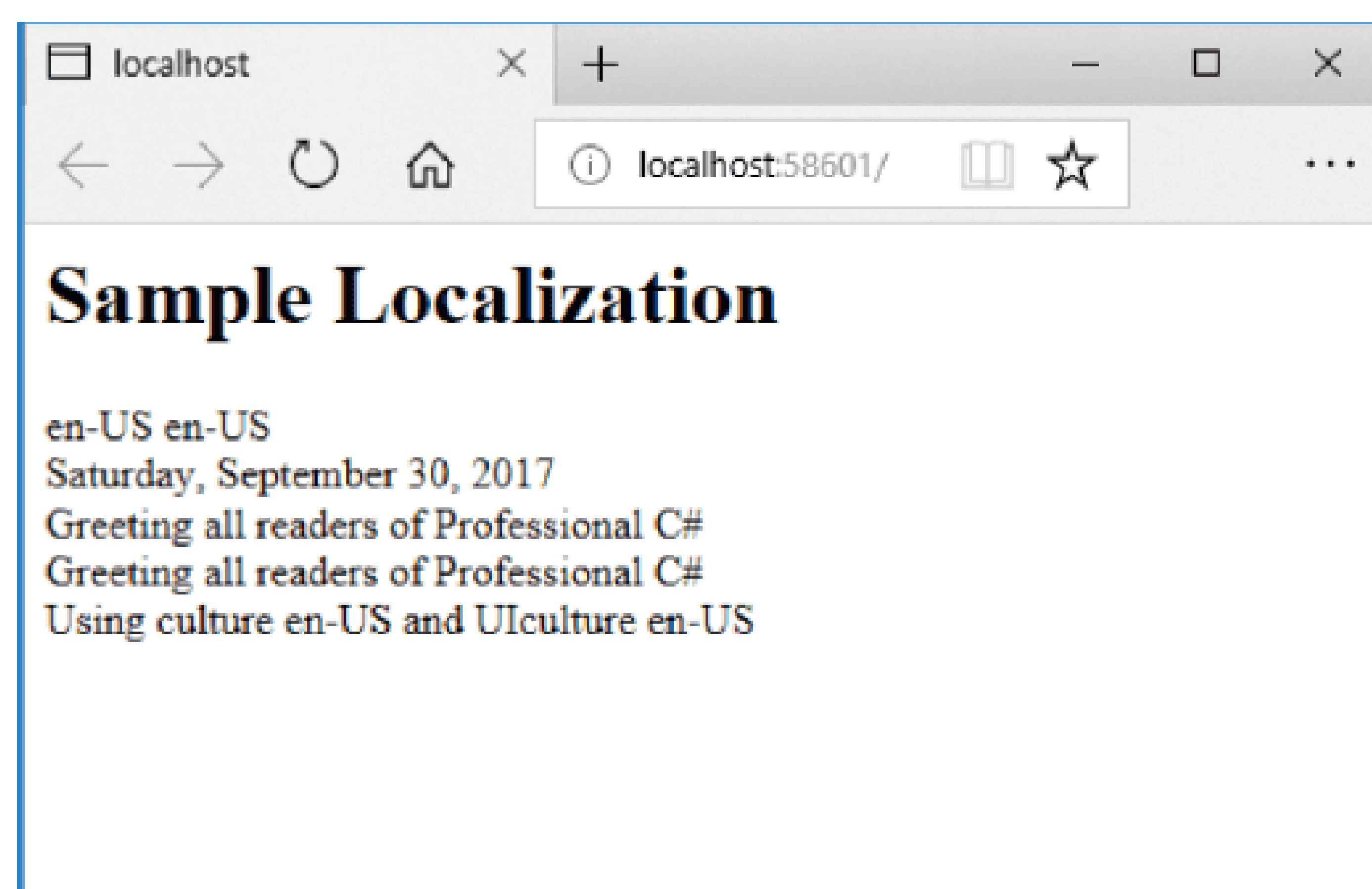



图 27-7

给 URL 请求添加?culture=de-AT(它使用了 QueryStringRequestCultureProvider)，输出就如图 27-8 所示。传递 URL 请求不支持的区域性，输出就使用默认的区域性。




图 27-8

### 27.4.5 使用控制器和视图进行本地化

使用 ASP.NET Core MVC，有对本地化的特殊支持。可以为控制器和视图创建特定的资源文件，可以向用于从资源中检索值的模型数据添加注释。

**注意：**
ASP.NET Core MVC 详见第 31 章。注释详见第 16 章。

本节使用的 ASP.NET Core Web 应用程序 ASPNETCoreMVCSample 基于 Web Application (Model View Controller)模板，使用以下依赖项和名称空间：

**依赖项**

Microsoft.AspNetCore.All

**名称空间**

Microsoft.AspNetCore
Microsoft.AspNetCore.Builder
Microsoft.AspNetCore.Hosting
Microsoft.AspNetCore.Localization
Microsoft.AspNetCore.Mvc.Localization
Microsoft.AspNetCore.Mvc.Razor
Microsoft.Extensions.Configuration
Microsoft.Extensions.DependencyInjection
Microsoft.Extensions.Localization
System
System.ComponentModel
System.Diagnostics
System.Globalization

下面从创建资源开始。资源现在存储在 Resources 文件夹中。使用 AddLocalization 辅助方法的选项指定文件夹名(代码文件 ASPNETCoreMVCSample/Startup.cs)：

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddLocalization(options => options.ResourcesPath = "Resources");
  //...
}
```

可以将控制器、视图和模型的资源放在子文件夹中，也可以使用文件名的点表示法。例如，可以将 HomeController 的资源放入子目录 Resources\Controllers——例如，使用默认语言的资源文件 HomeController.resx，以及特定于语言的资源文件(如 HomeController.de.resx)。不使用目录结构，而可以使用文件名的点表示法。这里，将资源文件直接存储在 Resources 文件夹中。使用这个约定，HomeController 的资源文件需要命名为 Controllers.HomeController.resx。

对于从 HomeController 中激活的 Hello 视图，可以对资源文件 Hello.resx 使用目录表示法，将其放入目录结构 Resources/Views/Home。使用点表示法，资源文件 Views.Home.Hello.resx 需要保存在文件夹 Resources 中。

使用的资源文件版本由 AddViewLocalization 方法指定，AddViewLocalization 方法是 IMvcBuilder 接口的扩展方法。实现此接口的对象从 AddMvc 方法返回，因此可以使用流利 API 语法调用 AddViewLocalization。传递选项 LanguageViewLocationExpanderFormat. SubFolder 时，它指定使用子文件夹资源。另一个选项是 LanguageViewLocationExpanderFormat.Suffix。在下面的代码片段中，还调用扩展方法 AddDataAnnotationsLocalization 来启用数据注释的本地化(代码文件 ASPNETCoreMVCSample/Startup.cs)：

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddLocalization(options => options.ResourcesPath = "Resources");
  services.AddMvc()
    .AddViewLocalization(LanguageViewLocationExpanderFormat.SubFolder)
    .AddDataAnnotationsLocalization();
}
```

使用 HomeController，可以注入 IStringLocalizer，就像在 WebSampleApp 示例的 Configure 方法中也注入了

IStringLocalizer。在下面的代码片段中，将 IStringLocalizer<homecontroller>注入 HomeController 的构造函数中，并与 Hello 操作方法一起使用(代码文件 ASPNETCoreMVCSample/Controllers/HomeController.cs)：

```
private readonly IStringLocalizer<HomeController> _localizer;
public HomeController(IStringLocalizer<HomeController> localizer)
{
  _localizer = localizer;
}

public IActionResult Hello()
{
  ViewBag.Message1 = _localizer.GetString("Message1");
  return View();
}
```

在视图中，直接访问从控制器传递的 Message1(代码文件 ASPNETCoreMVCSample/Views/Home/Hello.cshtml)：

```
<h3>@ViewBag.Message1</h3>
```

要在视图的资源中直接访问资源，可以使用@inject 声明注入 IViewLocalizer。这里定义了一个本地变量 Localizer，然后用于访问名为 Message2 的资源；资源只需要保存在资源文件 Resources/Views/Home/Hello.resx 中 (代码文件 ASPNETCoreMVCSample/Views/Home/Hello.cshtml)：

```
@using Microsoft.AspNetCore.Mvc.Localization

@inject IViewLocalizer Localizer

@{
    ViewData["Title"] = "Hello";
}

<h2>Hello</h2>

<h3>@ViewBag.Message1</h3>

<h3>@Localizer["Message2"]</h3>
```

运行应用程序，并访问链接/Home/Hello，从控制器和视图中获取资源，如图 27-9 所示。

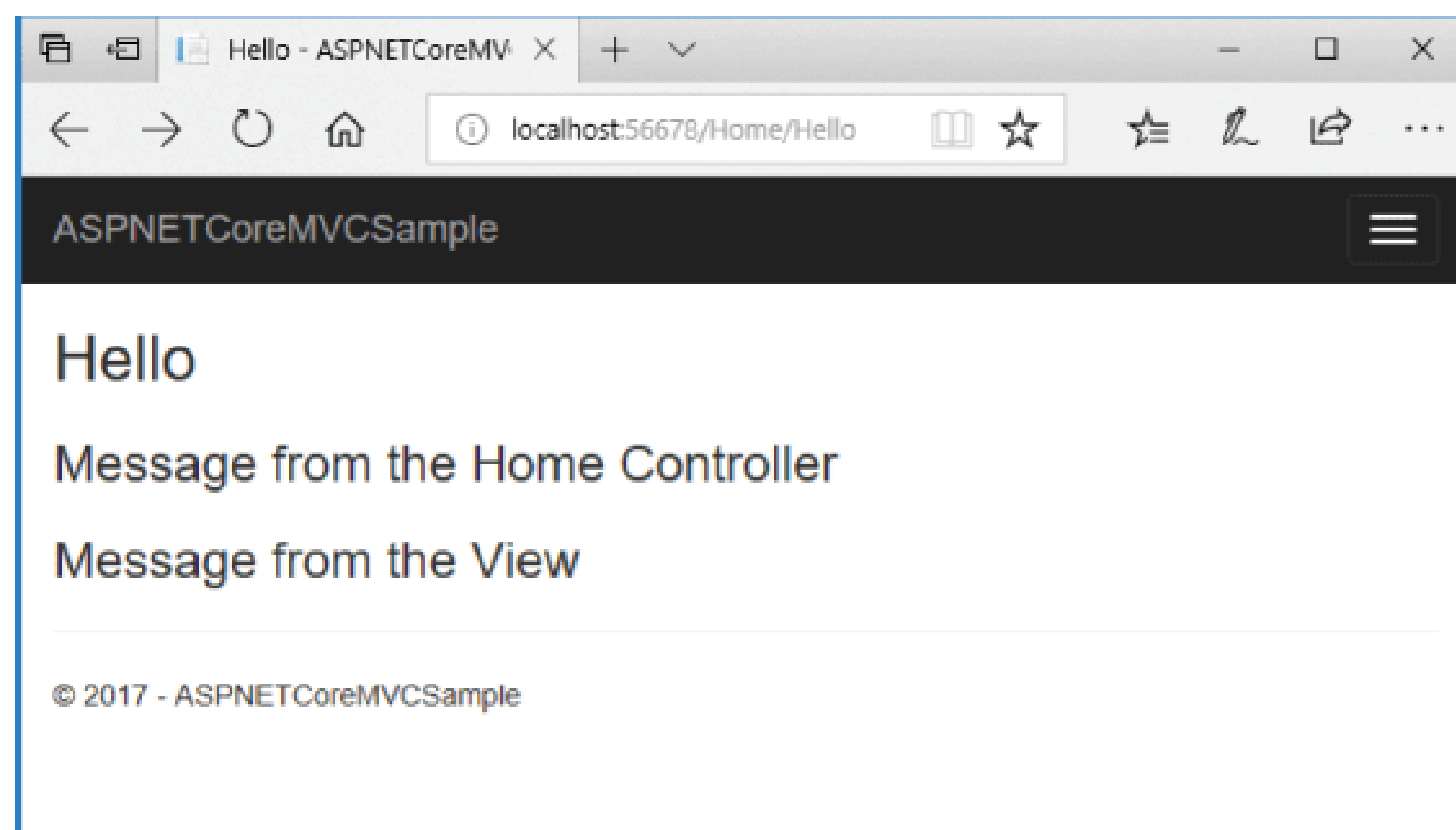



图 27-9

另一种使用 ASP.NET Core MVC 检索资源值的方法是通过应用注解实现的。要查看操作中的注释，可以在 Models 目录中定义 Book 类型。这种类型把 DisplayName 特性添加到属性 Booktitle 和 Publisher 中 (代码文件 ASPNETCoreMVCSample/Models/Book.cs)：

```
public class Book
{
  [DisplayName("Booktitle")]
  public string Booktitle { get; set; }
  [DisplayName("Publisher")]
  public string Publisher { get; set; }
}
```

**注意：**
不要忘记通过调用 Startup 类中的方法 AddDataAnnotationsLocalization 来启用注释的本地化。

资源文件 Book.resx 现在添加到目录 Resources/Models 中(资源文件 ASPNETCoreMVCSample/Resources/Models/Book.resx)：

```
<!-- ... -->
<data name="Publisher" xml:space="preserve">
  <value>Publisher</value>
</data>
<data name="Booktitle" xml:space="preserve">
  <value>Title</value>
</data>
<!-- ... -->
```

还添加了 German 语言表示(资源文件 ASPNETCoreMVCSample/Resources/Models/Book.de.resx)：

```
<!-- ... -->
<data name="Publisher" xml:space="preserve">
  <value>Verlag</value>
</data>
<data name="Booktitle" xml:space="preserve">
  <value>Titel</value>
</data>
<!-- ... -->
```

HomeController 中的 Book 操作方法将 Book 模型传递给视图(代码文件 ASPNETCoreMVCSample/Controllers/HomeController.cs)：

```
public IActionResult Book()
{
  var b = new Book
  {
    Booktitle = "Professional C# 7 and .NET Core 2",
    Publisher = "Wrox Press"
  };
  return View(b);
}
```

要显示图书，使用 HTML Helper 方法 EditorForModel 来显示带有输入字段的模型的所有属性(代码文件 ASPNETCoreMVCSample/Views/Home/Book.cshtml)：

```
@{
    ViewData["Title"] = "Book";
}

<h2>Book</h2>

@Html.EditorForModel()
```

图 27-10 显示正在运行的应用程序用 German 区域性访问 Home/Book。

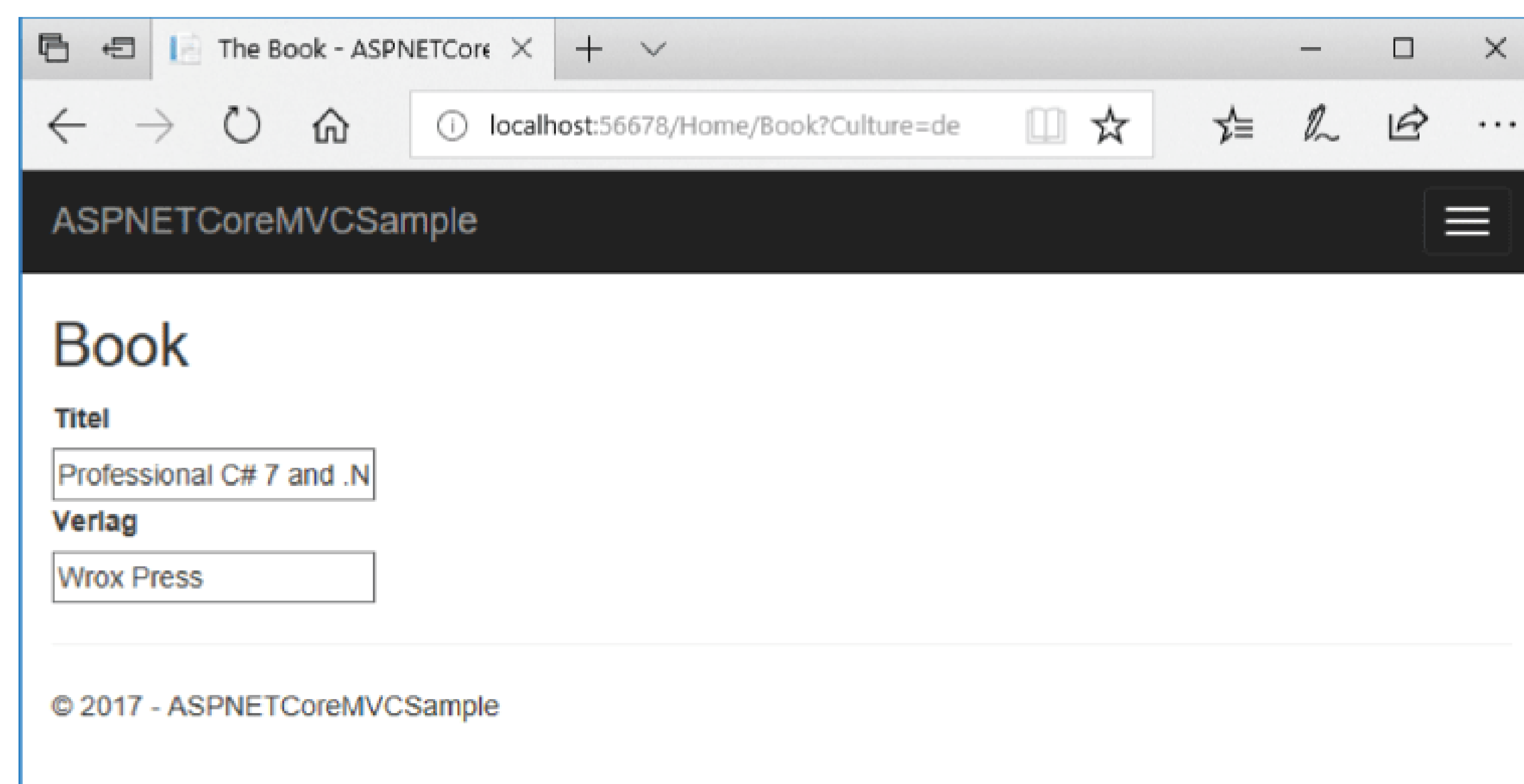



图 27-10

**注意：**
控制器、视图和 HTML Helper 详见第 31 章。

# 27.5 本地化 UWP

用 Universal Windows Platform (UWP)进行本地化基于前面学习的概念，但带来了一些新理念，如下所述。为了获得最佳的体验，需要通过 Visual Studio Extensions and Updates 安装 Multilingual App Toolkit。

区域性、区域和资源的概念是相同的，但因为 Windows 应用程序可以用 C#和 XAML、C++和 XAML、JavaScript 和 HTML 来编写，所以这些概念必须可用于所有的语言。只有 Windows Runtime 能用于所有这些编程语言和 UWP 应用程序。因此，用于全球化和资源的新名称空间可通过 Windows Runtime 来使用：Windows.Globalization 和 Windows.AppalicationModel.Resources。在全球化名称空间中包含 Calendar、GeographicRegion(对应于.NET 的 RegionInfo)和 Language 类。在其子名称空间中，还有一些数字和日期格式化类随着语言的不同而改变。在 C#和 Windows 应用程序中，仍可以使用.NET 类表示区域性和区域。

下面举一个例子，说明如何用 Universal Windows 应用程序进行本地化。使用 Blank App (Universal Windows)Visual Studio 项目模板创建一个小应用程序。在页面上添加两个 TextBlock 控件和一个 TextBox 控件。

在代码文件的 OnNavigatedTo()方法中，可以把具有当前格式的日期赋予 text1 控件的 Text 属性。DateTime 结构可以用非常类似于本章前面控制台应用程序的方式使用(代码文件 UWPLocalization/MainPage.xaml.cs)：

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
  base.OnNavigatedTo(e);
  text1.Text = DateTime.Today.ToString("D");
  //...
}
```

## 27.5.1 给 UWP 使用资源

在 UWP 中，可以用文件扩展名 resw 替代 resx，以创建资源文件。在后台，resw 文件使用相同的 XML 格式，可以使用相同的 Visual Studio 资源编辑器创建和修改这些文件。下例使用如图 27-11 所示的结构。子文件夹 Message 包含一个子目录 en-us，在其中创建了两个资源文件 Errors.resw 和 Messages.resw。在 Strings\en-us 文件夹中，创建了资源文件 Resources.resw。

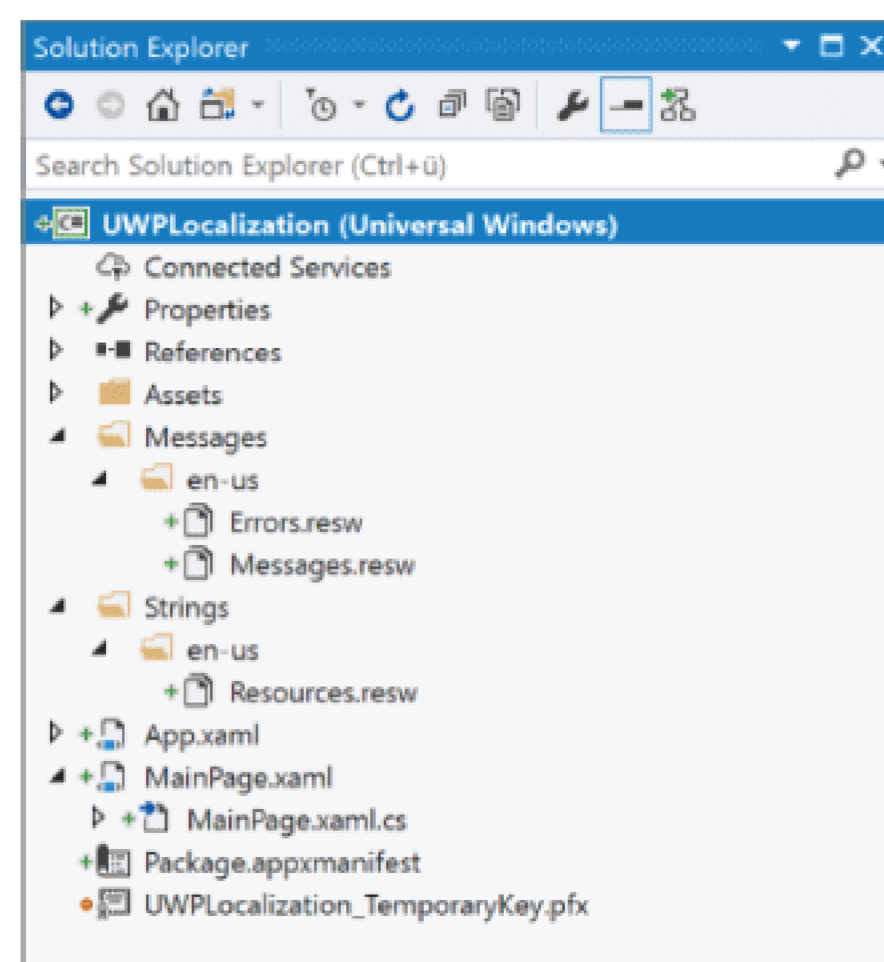


图 27-11

Messages.resw 文件包含一些英语文本资源，Hello 的值是 Hello World，资源的名称是 GoodDay、GoodEvening 和 GoodMorning。文件 Resources.resw 包含资源 Text3.Text 和 Text3.Width，其值分别是“This is a sample message for Text 4”和 300。

在代码中，使用 Windows.AppalicationModel.Resources 名称空间中的 ResourcesLoader 类可以访问资源。这里使用字符串“Messages”作为 GetForCurrentView 方法的参数。因此，要使用资源文件 Messages.resw。调用 GetString 方法，会检索键为“Hello”的资源(代码文件 UWPLocalization/MainPage.xaml.cs)。

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
  //...
  var resourceLoader = ResourceLoader.GetForCurrentView("Messages");
  text2.Text = resourceLoader.GetString("Hello");
}
```

在 UWP Windows 应用程序中，也可以直接在 XAML 代码中使用资源。对于下面的 TextBox，给 x:Uid 特性赋予值 Text3。这样，就会在资源文件 Resources.resw 中搜索名为 Text3 的资源。这个资源文件包含键 Text3.Text 和 Text3.Width 的值。检索这些值，并设置 Text 和 Width 属性(代码文件 UWPLocalization/MainPage.xaml)：

```
<TextBox x:Uid="FileName_Text3" HorizontalAlignment="Left" Margin="40"
  TextWrapping="Wrap" Text="TextBox" VerticalAlignment="Top"/>
```

在这个阶段运行应用程序，可以看到资源文件中的英语文本，如图 27-12 所示。

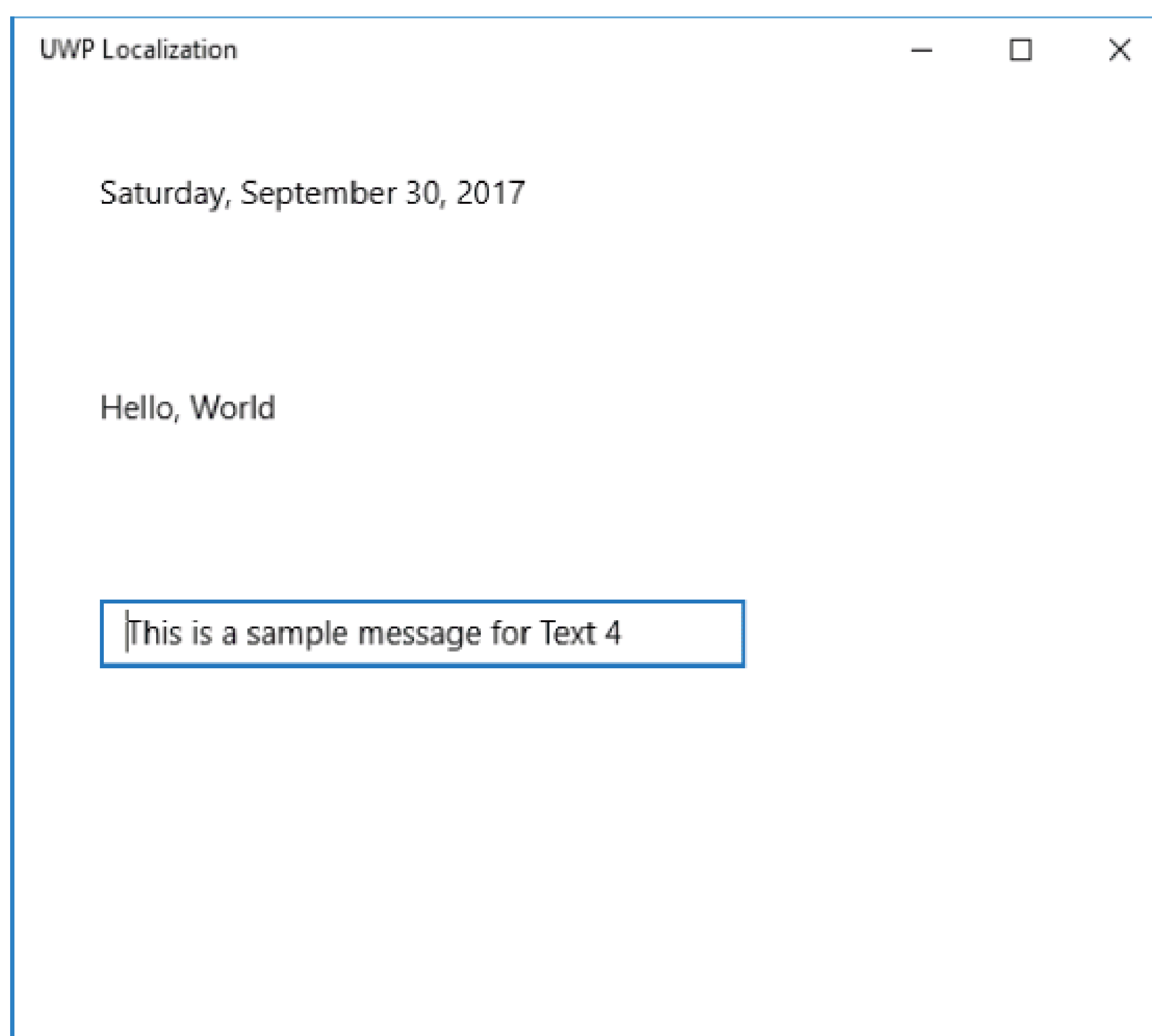



图 27-12

## 27.5.2　使用多语言应用程序工具集进行本地化

为了本地化 UWP 应用程序，可以下载 Multilingual App Toolkit。这个工具包集成在 Visual Studio 2017 中。安装了该工具包后，就可以通过 Tools | Multilingual App Toolkit | Enable Selection 菜单，为 UWP 应用程序启用它。这会在项目文件中添加一个生成命令，在 Solution Explorer 的上下文菜单中添加一个菜单项。打开该上下文菜单，选择 Multilingual App Toolkit | Add Translation Languages，打开如图 27-13 所示的对话框，在其中可以选择要翻译为哪种语言。该示例选择 Pseudo Language、French、German 和 Spanish。对于这些语言，可以使用 Microsoft Translator。这个工具现在创建一个 MultilingualResources 子目录，其中包含所选语言的.xlf 文件。.xlf 文件用 XLIFF(XML Localization Interchange File Format，XML 本地化数据交换格式)标准定义，这是 Open Architecture for XML Authoring and Localization(OAXAL)参考架构的一个标准。

**注意：**

*Multilingual App Toolkit 也可以在 http://aka.ms/ matinstallv4 上安装，不需要使用 Visual Studio。下载 Multilingual App Toolkit。*

下次启动项目的生成过程时，XLIFF 文件就会从所有资源中填充相应的内容。在 Solution Explorer 中选择 XLIFF 文件，就可以把它们直接发送给翻译过程。为此，在 Solution Explorer 中打开上下文菜单，选择.xlf 文件，选择 Multilingual App Toolkit | Export Translations，打开如图 27-14 所示的对话框，在其中可以配置应发送的信息，也可以发送电子邮件，添加 XLIFF 文件作为附件。

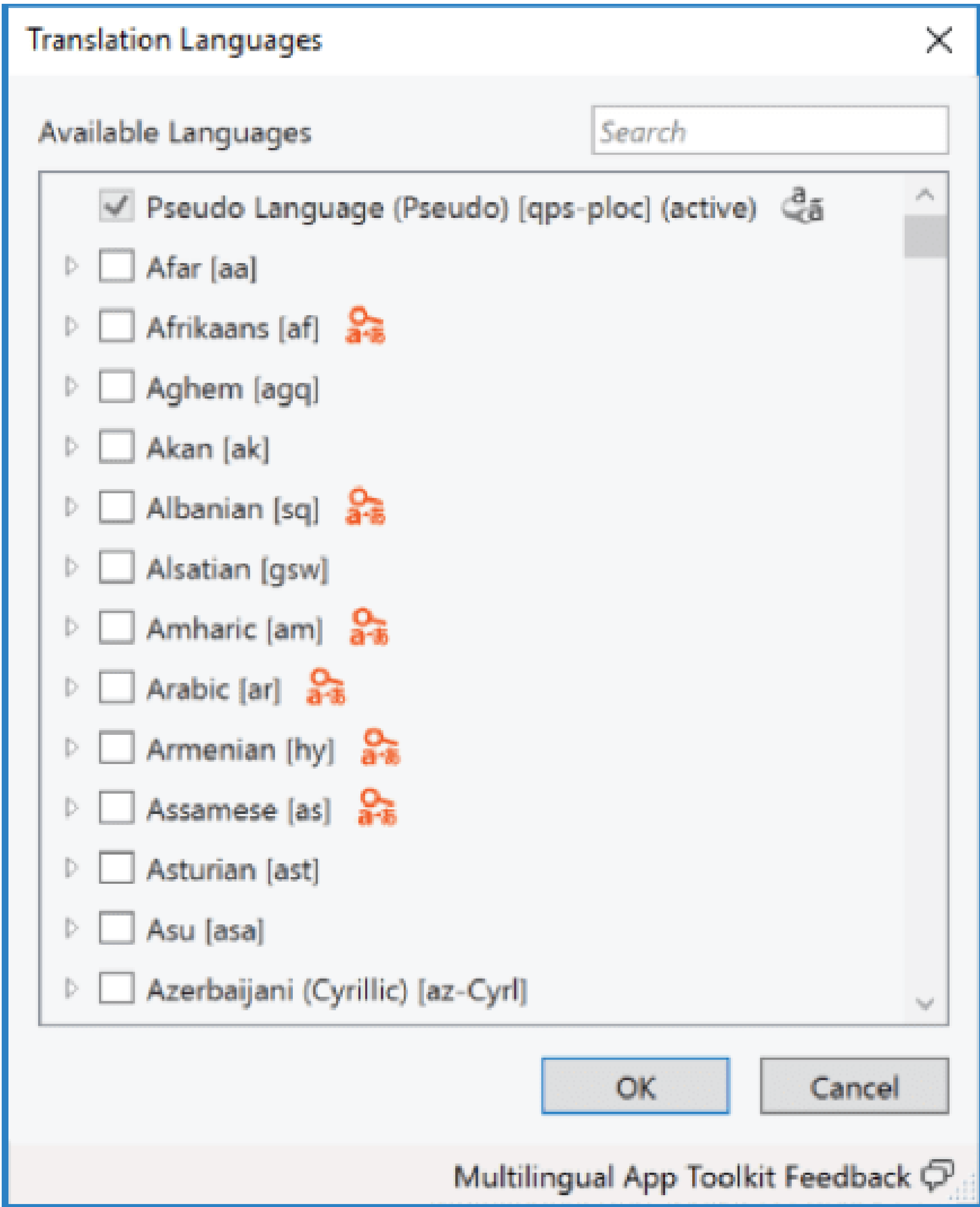



图 27-13

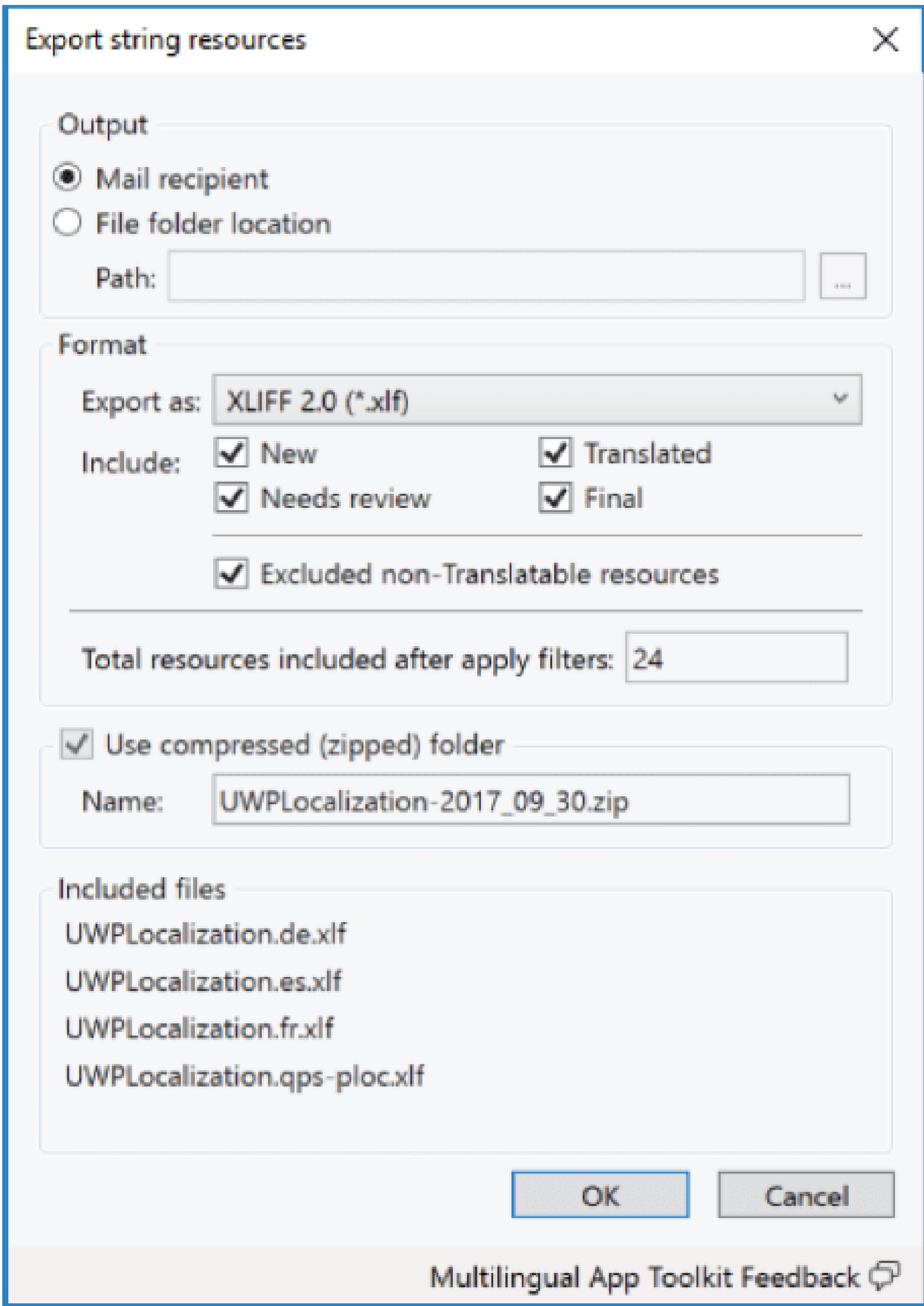



图 27-14

对于翻译，还可以使用微软的翻译服务。在 Visual Studio Solution Explorer 中选择.xlf 文件，打开上下文菜单后，选择 Multilingual App Toolkit | Generate Machine Translations。为了使这个操作可用于本书撰写时的版本，需要创建一个 Microsoft Azure 账户，添加 Translator Speech API of Microsoft Cognitive Services。这个服务由 Multilingual App Toolkit 的翻译服务使用。还有免费的服务，允许每个月翻译 2 000 000 个字符。

打开.xlf 文件时，会打开 Multilingual Editor (参见图 27-15)。有了这个工具，就可以验证自动翻译，并进行必要的修改。

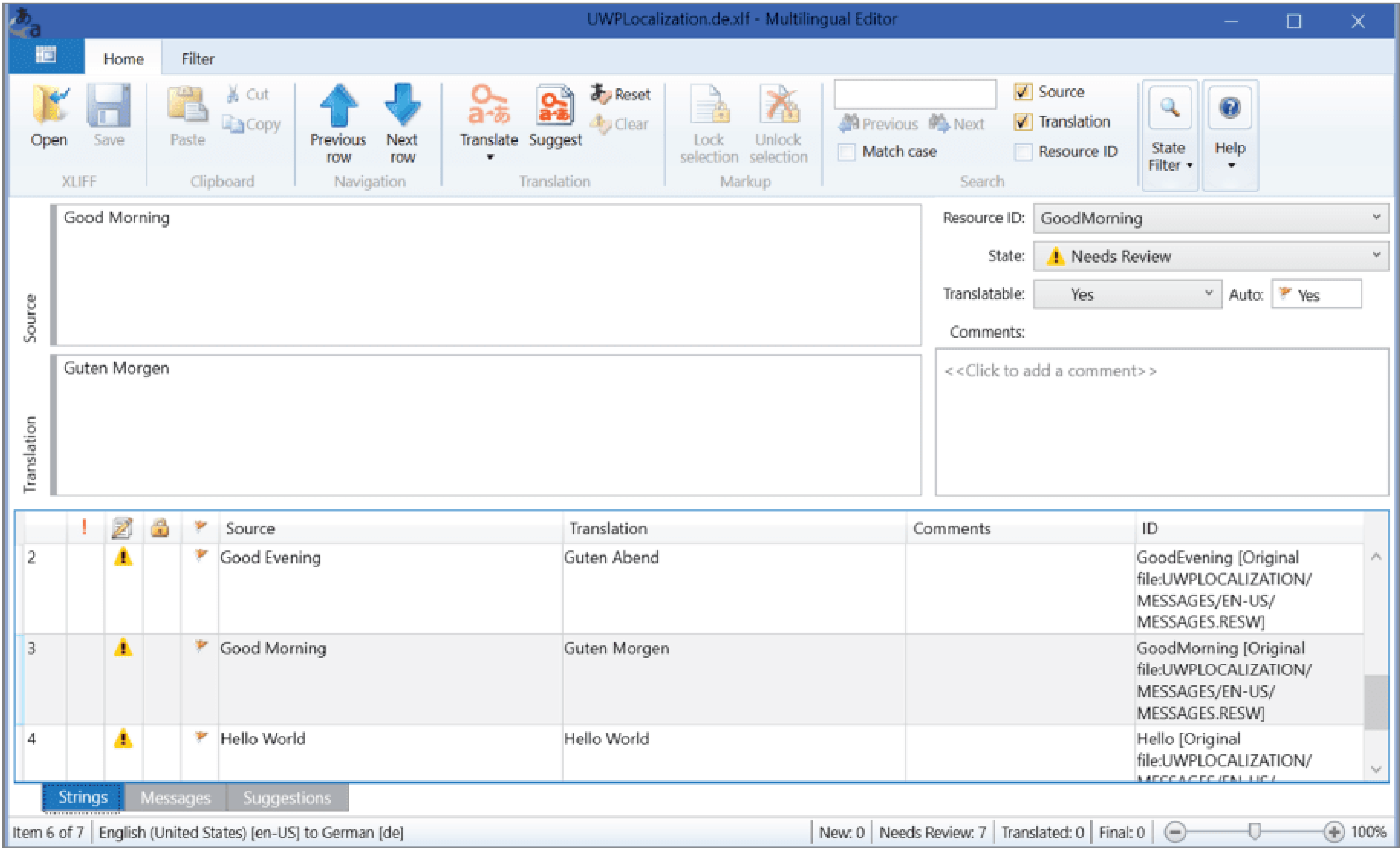



图 27-15

**注意：**

没有人工检查，就不要使用机器翻译。该工具会为每个已翻译的资源显示状态。自动翻译完成后，状态设置为 Needs Review。自动翻译的结果可能不正确，有时还很可笑。

## 27.6 小结

本章讨论了.NET 应用程序的全球化和本地化。对于应用程序的全球化，我们讨论了 System.Globalization 名称空间，它用于格式化依赖于区域性的数字和日期。此外，说明了在默认情况下，字符串的排序取决于区域性。我们使用不变的区域性进行独立于区域性的排序。

应用程序的本地化使用资源来实现。资源可以放在文件或附属程序集中。本地化所使用的类位于 System.Resources 名称空间中。

我们还学习了如何本地化 ASP.NET Core 应用程序，给 ASP.NET Core MVC 使用特殊的功能以及使用 UWP 的本地化应用程序。

下一章介绍测试，学习如何创建单元测试，以及用于 Windows 和 Web 应用程序的 UI 测试。

# 第28章

# 测　试

**本章要点**

- 使用 MSTest 和 xUnit 的单元测试
- 使用 xUnit 和.NET Core 的单元测试
- 使用 EF Core 的单元测试
- 编码的 UI 测试
- Web 测试

**本章源代码下载：**

打开 www.wrox.com 的 Download Code 选项卡可下载本章源代码。源代码也可以在 tests 目录的 https://github.com/ProfessionalCSharp/ProfessionalCSharp7 中找到。本章代码分为如下主要示例：

- Unit Testing Sample
- Mocking Sample
- EF Core Sample
- Windows App Sample
- ASPNETCore Integration Test Sample
- Web Application Load Test Sample

**注意：**

单元测试可用于 Visual Studio 的所有版本，其他所有的测试功能，例如实时单元测试、Web 负载和性能测试、编码的 UI 测试、测试覆盖、Microsoft Fakes 和 IntelliTest 都需要 Visual Studio 2017 企业版。

## 28.1 概述

应用程序开发正在变得敏捷。使用瀑布过程模型来分析需求时，设计应用程序架构，实现它，两三年后发现所建立的应用程序没有满足用户的需求，这种情形并不少见。相反，软件开发变得敏捷，发布周期更短，最终用户在开发早期就参与进来。看看 Windows 10：数以百万计的 Windows 内部人士给早期的构建版本提供反馈，每隔几个月甚至几周就更新一次。在 Windows 10 的 Beta 程序中，Windows 内部人士曾经在一周内收到

Windows 10 的 3 个构建版本。Windows 10 是一个巨大的程序，但微软设法在很大程度上改变开发方式。同样，如果参与.NET Core 开源项目，每晚都会收到 NuGet 包的构建版本。如果喜欢冒险，甚至可以写一本关于未来技术的书。

如此快速和持续的改变——每晚都创建构建版本——等不及内部人士或最终用户发现所有问题。Windows 10 每隔几分钟就崩溃一次，Windows 10 内部人士就不会满意。修改方法的实现代码的频率是多少，才能发现似乎不相关的代码不工作了？为了试图避免这样的问题，不改变方法，而是创建一个新的方法，复制原来的代码，并进行必要的修改，但这将极难维护。在一个地方修复方法后，太容易忘记修改其他方法中重复的代码。

为了避免这样的问题，可以给方法创建测试程序，使测试程序自动运行，签入源代码或在每晚的构建过程中检查。从一开始就创建测试程序，会在开始时增加项目的成本，但随着项目的继续进行和维护期间，创建测试程序有其优点，降低了项目的整体成本。

本章解释了各种各样的测试，从测试小功能的单元测试开始。这些测试应该验证应用程序中可测试的最小部分的功能，例如方法。传入不同的输入值时，单元测试应该检查方法的所有可能路径。

MSTest 是 Visual Studio 用于创建单元测试的测试框架。建立.NET Core 时，MSTest 不支持为.NET Core 库和应用程序创建测试。这就是为什么微软使用 xUnit 为.NET Core 创建单元测试的原因。现在可以使用 MSTest 和 xUnit 为.NET Core 创建单元测试。本章介绍微软的测试框架 MSTest 和 xUnit。

使用 Web 测试，可以测试 Web 应用程序，发送 HTTP 请求，模拟一群用户。创建这些类型的测试，允许模拟不同的用户负载，允许进行压力测试。可以使用测试控制器，来创建更高的负载，模拟成千上万的用户，从而也知道需要什么基础设施，应用程序是否可伸缩。

本章介绍的最后一个测试特性是 UI 测试。可以为基于 XAML 的应用程序创建自动化测试。当然，更容易为视图模型创建单元测试，用 ASP.NET Core 创建视图组件，但本章不可能涉及测试的方方面面。可以自动化 UI 测试。想象一下数百种不同的 Android 移动设备。你会购买每一个型号，在每个设备上手动测试应用程序吗？最好使用云服务，在确实要安装应用程序的、数以百计的设备上，发送要测试的应用程序。不要以为人们会在数以百计的设备上启动云中的应用程序，并与应用程序进行可能的交互，这需要使用 UI 测试自动完成。

首先，创建单元测试。

## 28.2　使用 MSTest 进行单元测试

编写单元测试有助于代码维护。例如，在更新代码时，想要确信更新不会破坏其他代码。创建自动单元测试可以帮助确保修改代码后，所有功能得以保留。Visual Studio 2017 提供了一个健壮的单元测试框架，还可以在 Visual Studio 内使用其他测试框架。

### 28.2.1　使用 MSTest 创建单元测试

下面的示例测试类库 UnitTestingSamples 中一个非常简单的方法。这是一个.NET 标准 2.0 类库。当然，可以创建其他基于 MSBuild 的项目。类 DeepThought 包含 TheAnswerToTheUltimateQuestionOfLifeTheUniverse-AndEverything 方法，该方法返回 42 作为结果(代码文件 UnitTestingSamples/DeepThought.cs)：

```
public class DeepThought
{
  public int TheAnswerOfTheUltimateQuestionOfLifeTheUniverseAndEverything() =>
    42;
}
```

为了确保没有人改变返回错误结果的方法，创建一个单元测试。要创建单元测试，可以使用 dotnet 命令：

```
> dotnet new mstest
```

也可以使用 Visual Studio 中的 Unit Test Project (.NET Core)项目模板。

开始创建第一个测试之前，最好考虑一下测试和测试项目的命名。当然，可以使用任何名称，但.NET Core 团队提供了较好的命名规则，参阅：

https://github.com/aspnet/Home/wiki/Engineering-guidelines#unit-tests-and-functional-tests

下面是规则汇总：

- 测试项目的名称是在项目名后加上 Tests，例如，对于项目 UnitTestingSamples，测试项目的名称是 UnitTestingSamples.Tests。
- 测试类名与被测试的类名相同，后跟 Test，例如，UnitTestingSamples.DeepThought 的测试类是 UnitTestingSamples.DeepThoughtTest。
- 单元测试名采用描述性的名称，例如，名称 AddOrUpdateBookAsync_ ThrowsForNull 表示，一个单元测试调用 AddOrUpdateBookAsync 方法，检查传递 null 时它是否抛出异常。

MSTest 项目包含对 NuGet 包 Microsoft.NET.Test.Sdk、MSTest.TestAdapter 和 MSTest.TestFramework 的引用(项目文件 UnitTestingSamples.MSTests/UnitTestingSamples.MSTests.csproj)：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
    <IsPackable>false</IsPackable>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.NET.Test.Sdk" Version="15.5.0" />
    <PackageReference Include="MSTest.TestAdapter" Version="1.2.0" />
    <PackageReference Include="MSTest.TestFramework" Version="1.2.0" />
  </ItemGroup>

  <ItemGroup>
    <ProjectReference Include="..\UnitTestingSamples\UnitTestingSamples.csproj" />
  </ItemGroup>

</Project>
```

单元测试类标有 TestClass 特性，测试方法标有 TestMethod 特性。该实现方式创建 DeepThought 的一个实例，并调用要测试的方法 TheAnswerToTheUltimateQuestionOfLifeTheUniverseAndEverything。返回值使用 Assert.AreEqual 与 42 进行比较。如果 Assert.AreEqual 失败，测试就失败(代码文件 UnitTestingSamples.MSTests/DeepThoughtTests.cs)：

```
[TestClass]
public class TestProgram
{
  [TestMethod]
  public void
    ResultOfTheAnswerToTheUltimateQuestionOfLifeTheUniverseAndEverything()
  {
    // arrange
    int expected = 42;
    var dt = new DeepThought();

    // act
    int actual =
      dt.TheAnswerToTheUltimateQuestionOfLifeTheUniverseAndEverything();

    // assert
    Assert.AreEqual(expected, actual);
  }
}
```

单元测试由 3 个 A 定义：Arrange、Act 和 Assert。首先，一切都安排好了，单元测试可以开始了。在安排阶段，在第一个测试中，给变量 expected 分配调用要测试的方法时预期的值，调用 DeepThought 类的一个实例。现在准备好测试功能了。在行动阶段，调用方法。在完成行动阶段后，需要验证结果是否与预期相同。这在断言阶段使用 Assert 类的方法来完成。

Assert 类是 Microsoft.VisualStudio.TestTools.UnitTesting 名称空间中 MSTest 框架的一部分。这个类提供了一些可用于单元测试的静态方法。默认情况下，Assert.Fail 方法添加到自动创建的单元测试中，提供测试还没有实现的信息。其他一些方法有：AreNotEqual 验证两个对象是否不同；IsFalse 和 IsTrue 验证布尔结果；IsNull 和 IsNotNull 验证空结果；IsInstanceOfType 和 IsNotInstanceOfType 验证传入的类型。

### 28.2.2　运行单元测试

使用 Test Explorer(通过 Test | Windows | Test Explorer 打开)，可以在解决方案中运行测试(见图 28-1)。

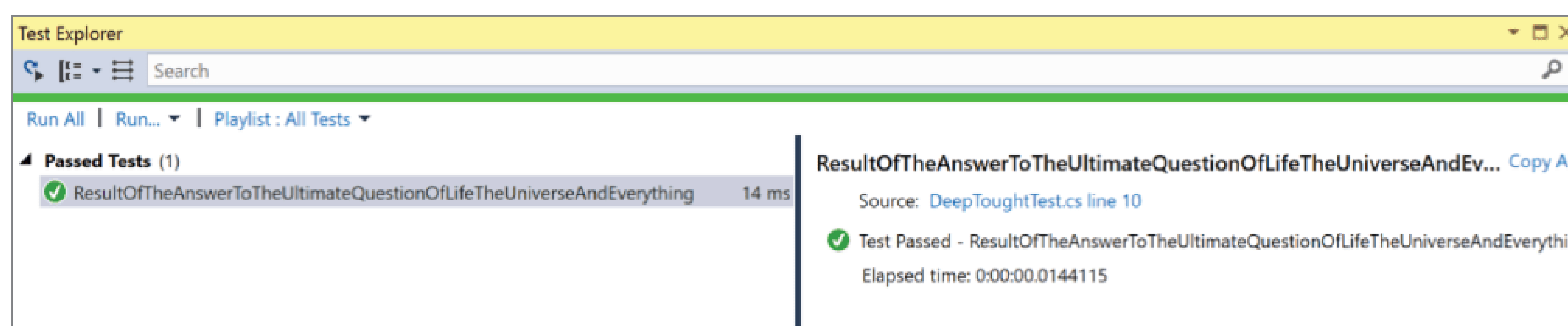



图 28-1

图 28-2 显示了一个失败的测试，列出了失败的所有细节。

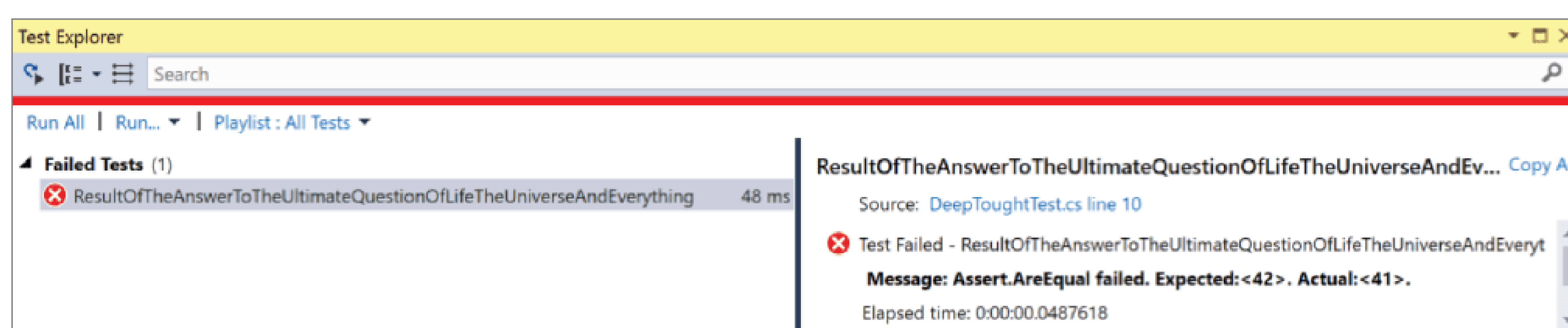



图 28-2

要在命令行上运行测试，可以调用 dotnet test：

```
> dotnet test
```

在示例应用程序中，会得到成功的结果：

```
Build started, please wait...
Build completed.

Test run for
C:\procsharp\Tests\UnitTestingSamples\UnitTestingSamples.MSTests\bin\Debug\
netcoreapp2.0\UnitTestingSamples.MSTests.dll(.NETCoreApp,Version=v2.0)
Microsoft (R) Test Execution Command Line Tool Version 15.5.0
Copyright (c) Microsoft Corporation.  All rights reserved.

Starting test execution, please wait...

Total tests: 1. Passed: 1. Failed: 0. Skipped: 0.
Test Run Successful.
Test execution time: 1.2312 Seconds
```

当然，这只是一个很简单的场景，测试通常是没有这么简单的。例如，方法可以抛出异常，用其他的路径返回其他值，或者使用了不应该在单个单元中测试的代码(例如数据库访问代码或者调用的服务)。接下来介绍一个比较复杂的单元测试场景。

下面的类 StringSample 定义了一个带字符串参数的构造函数、方法 GetStringDemo 和一个字段。方法 GetStringDemo 根据 first 和 second 参数使用不同的路径，并返回一个从这些参数得到的字符串(代码文件 UnitTestingSamples /StringSample.cs)：

```
public class StringSample
{
  public StringSample(string init)
  {
    if (init is null)
      throw new ArgumentNullException(nameof(init));

    _init = init;
  }

  private string _init;
```

```
public string GetStringDemo(string first, string second)
{
  if (first is null)
  {
    throw new ArgumentNullException(nameof(first));
  }
  if (string.IsNullOrEmpty(first))
  {
    throw new ArgumentException("empty string is not allowed", first);
  }
  if (second is null)
  {
    throw new ArgumentNullException(nameof(second));
  }
  if (second.Length > first.Length)
  {
    throw new ArgumentOutOfRangeException(nameof(second),
      "must be shorter than first");
  }
  int startIndex = first.IndexOf(second);
  if (startIndex < 0)
  {
    return $"{second} not found in {first}";
  }
  else if (startIndex < 5)
  {
    string result = first.Remove(startIndex, second.Length);
    return $"removed {second} from {first}: {result}";
  }
  else
  {
    return _init.ToUpperInvariant();
  }
}
}
```

**注意：**

为复杂的方法编写单元测试时，有时单元测试也会变得复杂起来。这有助于调试单元测试，找出当前执行的操作。调试单元测试很简单：给单元测试代码添加断点，并从 Test Explorer 的上下文菜单中选择 Debug Selected Tests (参见图 28-3)。

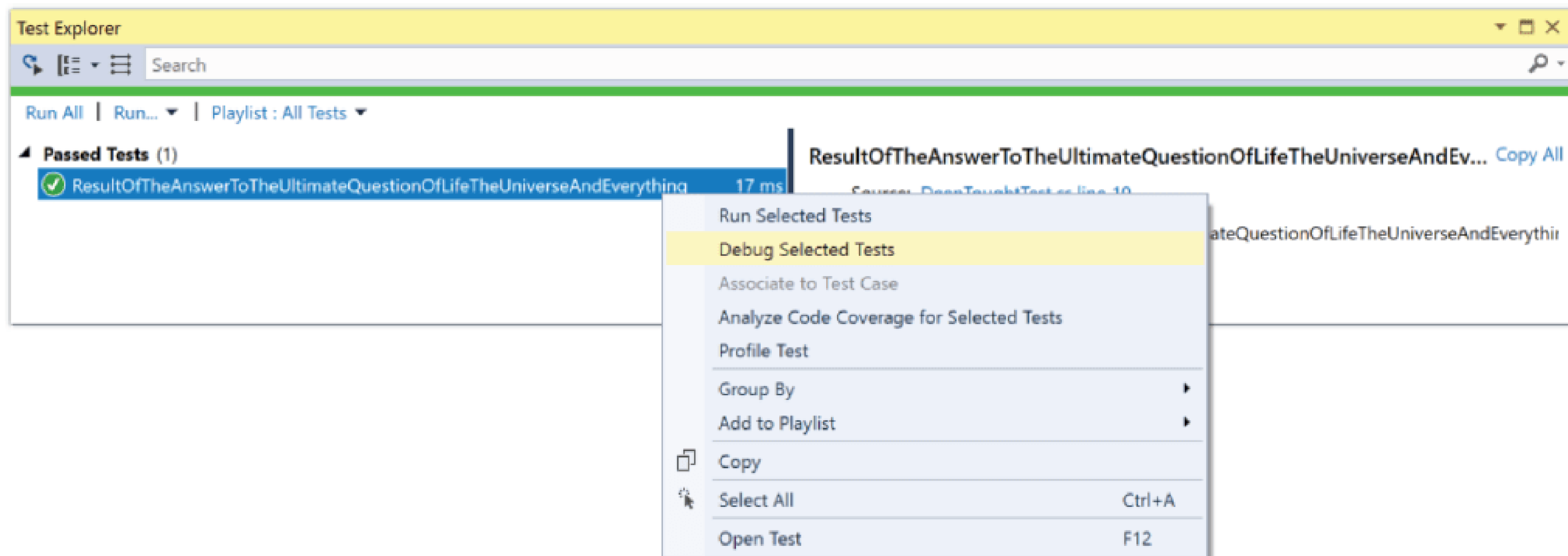



图 28-3

单元测试应该测试每个可能的执行路径，并检查异常。

## 28.2.3 使用 MSTest 预期异常

以 null 为参数调用 StringSample 类的构造函数和 GetStringDemo 方法时，可以预计会发生 ArgumentNullException 异常。在测试代码中很容易测试这一点，只需要像下面的示例那样对测试方法应用 ExpectedException 特性。这样，测试方法将成功地捕捉到异常(代码文件 UnitTestingSamples.MSTests/

StringSampleTests.cs)：

```
[TestMethod]
[ExpectedException(typeof(ArgumentNullException))]
public void ConstructorShouldThrowOnNull()
{
  var sample = new StringSample(null);
}
```

对于 GetStringDemo 方法抛出的异常，可以采取类似的处理。

### 28.2.4　测试全部代码路径

为了测试全部代码路径，可以创建多个测试，每个测试针对一条代码路径。下面的测试示例将字符串 a 和 b 传递给 GetStringDemo 方法。因为第二个字符串没有包含在第一个字符串内，所以 if 语句的第一个路径生效。结果将被相应地检查(代码文件 UnitTestingSamples.MSTests/StringSampleTests.cs)：

```
[TestMethod]
public void GetStringDemoBNotInA()
{
  string expected = "b not found in a";
  var sample = new StringSample(String.Empty);
  string actual = sample.GetStringDemo("a", "b");
  Assert.AreEqual(expected, actual);
}
```

下一个测试方法验证 GetStringDemo 方法的另一个路径。在这个示例中，第二个字符串包含在第一个字符串内，并且索引小于 5，所以将执行 if 语句的第二个代码块：

```
[TestMethod]
public void GetStringDemoRemoveBCFromABCD()
{
  string expected = "removed bc from abcd: ad";
  var sample = new StringSample(String.Empty);
  string actual = sample.GetStringDemo("abcd", "bc");
  Assert.AreEqual(expected, actual);
}
```

其他所有代码路径都可以以类似的方式测试。为了查看单元测试覆盖了哪些代码，以及还缺少什么代码，可以启动 Visual Studio 2017 中的 Code Coverage，使用 dotnet test 命令的--collect 选项。在 Visual Studio 2017 的 Code Coverage Results 窗口(Test | Windows | Code Coverage Results)中，可以看到单元测试覆盖代码的百分比。

### 28.2.5　外部依赖

许多方法都依赖于不受应用程序本身控制的某些功能，例如调用 Web 服务或者访问数据库。在测试外部资源的可用性时，可能服务或数据库并不可用。更糟的是，数据库和服务可能在不同的时间返回不同的数据，这就很难与预期的数据进行比较。在单元测试中，必须排除这种情况。

下面的示例依赖于外部的某些功能。方法 ChampionsByCountry()访问一个 Web 服务器上的 XML 文件，该文件以 Firstname、Lastname、Wins 和 Country 元素的形式列出了一级方程式世界冠军。这个列表按国家筛选，并使用 Wins 元素的值按数字顺序排序。返回的数据是一个 XElement，其中包含了转换后的 XML 代码：

```
public XElement ChampionsByCountry(string country)
{
  XElement champions = XElement.Load(F1Addresses.RacersUrl);
  var q = from r in champions.Elements("Racer")
          where r.Element("Country").Value == country
          orderby int.Parse(r.Element("Wins").Value) descending
          select new XElement("Racer",
            new XAttribute("Name", r.Element("Firstname").Value + " " +
              r.Element("Lastname").Value),
            new XAttribute("Country", r.Element("Country").Value),
            new XAttribute("Wins", r.Element("Wins").Value));
          return new XElement("Racers", q.ToArray());
}
```

**注意：**
关于 LINQ to XML 的更多信息，请参考网上附加第 2 章。

到 XML 文件的链接由 F1Addresses 类定义 (代码文件 UnitTestingSamples/F1Addresses.cs)：

```
public class F1Addresses
{
  public const string RacersUrl =
    "http://www.cninnovation.com/downloads/Racers.xml";
}
```

应该为 ChampionsByCountry 方法创建一个单元测试。测试不应依赖于服务器上的数据源。一方面，服务器可能不可用。另一方面，服务器上的数据可能随时间发生改变，返回新的冠军和其他值。正确的测试应该确保按预期方式完成筛选，并以正确的顺序返回正确筛选后的列表。

创建独立于数据源的单元测试的一种方法是使用依赖注入模式，重构 ChampionsByCountry 方法的实现代码。在这里，创建一个返回 XElement 的工厂，来取代 XElement.Load 方法。IChampionsLoader 接口是在 ChampionsByCountry 方法中使用的唯一外部要求。IChampionsLoader 接口定义了方法 LoadChampions，可以代替上述方法(代码文件 UnitTestingSamples /IChampionsLoader.cs)：

```
public interface IChampionsLoader
{
  XElement LoadChampions();
}
```

类 ChampionsLoader 使用 XElement.Load 方法实现了接口 IChampionsLoader，该方法由 ChampionsByCountry 方法预先使用(代码文件 UnitTestingSamples/ChampionsLoader.cs)：

```
public class ChampionsLoader: IChampionsLoader
{
  public XElement LoadChampions() => XElement.Load(F1Addresses.RacersUrl);
}
```

**注意：**
依赖注入模式参见第 20 章。

现在就能修改 ChampionsByCountry()方法的实现，使用接口而不是直接使用 XElement.Load 方法()来加载冠军。IChampionsLoader 传递给类 Formula1 的构造函数，然后 ChampionsByCountry()将使用这个加载器(代码文件 UnitTestingSamples/Formula1.cs)：

```
public class Formula1
{
  private IChampionsLoader _loader;
  public Formula1(IChampionsLoader loader)
  {
    _loader = loader;
  }

  public XElement ChampionsByCountry(string country)
  {
    var q = from r in _loader.LoadChampions().Elements("Racer")
            where r.Element("Country").Value == country
            orderby int.Parse(r.Element("Wins").Value) descending
            select new XElement("Racer",
              new XAttribute("Name", r.Element("Firstname").Value + " " +
                r.Element("Lastname").Value),
              new XAttribute("Country", r.Element("Country").Value),
              new XAttribute("Wins", r.Element("Wins").Value));
    return new XElement("Racers", q.ToArray());
  }
}
```

在典型的实现代码中，会把一个 ChampionsLoader 实例传递给 Formula1 构造函数，以从服务器检索赛车手。

创建单元测试时，可以实现一个自定义方法来返回一级方程式冠军，如方法 Formula1SampleData()所示(代码文件 UnitTestingSamples.MSTests/Formula1Tests.cs)：

```
internal static string Formula1SampleData()
```

```
{
  return @"
<Racers>
  <Racer>
    <Firstname>Nelson</Firstname>
    <Lastname>Piquet</Lastname>
    <Country>Brazil</Country>
    <Starts>204</Starts>
    <Wins>23</Wins>
  </Racer>
  <Racer>
    <Firstname>Ayrton</Firstname>
    <Lastname>Senna</Lastname>
    <Country>Brazil</Country>
    <Starts>161</Starts>
    <Wins>41</Wins>
  </Racer>
  <Racer>
    <Firstname>Nigel</Firstname>
    <Lastname>Mansell</Lastname>
    <Country>England</Country>
    <Starts>187</Starts>
    <Wins>31</Wins>
  </Racer>
  //... more sample data
```

方法 Formula1VerificationData 返回符合预期结果的样品测试数据(代码文件 UnitTestingSamples.MSTests/Formula1Test.cs)：

```
internal static XElement Formula1VerificationData()
{
  return XElement.Parse(@"
<Racers>
  <Racer Name=""Mika Hakkinen"" Country=""Finland"" Wins=""20"" />
  <Racer Name=""Kimi Raikkonen"" Country=""Finland"" Wins=""18"" />
</Racers>");
}
```

测试数据的加载器实现了与 ChampionsLoader 类相同的接口：IChampionsLoader。这个加载器仅使用样本数据，而不访问 Web 服务器(代码文件 UnitTestingSamples.MSTests/Formula1Test.cs)：

```
public class F1TestLoader: IChampionsLoader
{
  public XElement LoadChampions() => XElement.Parse(Formula1SampleData());
}
```

现在，很容易创建一个使用样本数据的单元测试(代码文件 UnitTestingSamples.MSTests/Formula1Test.cs)：

```
[TestMethod]
public void ChampionsByCountryFilterFinland()
{
  Formula1 f1 = new Formula1(new F1TestLoader());
  XElement actual = f1.ChampionsByCountry("Finland");
  Assert.AreEqual(Formula1VerificationData().ToString(), actual.ToString());
}
```

当然，真正的测试不应该只覆盖传递 Finland 作为一个字符串并在测试数据中返回两个冠军这样一种情况。还应该针对其他情况编写测试，例如传递没有匹配结果的字符串，返回两个以上的冠军的情况，可能还包括数字排序顺序与字母数字排序顺序不同的情况。

**注意：**

要测试不使用依赖注入的方法，用测试类替代在内部使用的依赖项，可以使用 Fakes Framework。这只能用于 Visual Studio 企业版的.NET Framework 项目。

# 28.3　使用 xUnit 进行单元测试

实现.NET Core 时，xUnit 可用于创建单元测试，.NET Core 团队使用了该产品。xUnit 是一个开源实现方案，创建 NUnit 2.0 的开发人员也创建了它。现在，.NET Core 命令行界面支持 MSTest 和 xUnit。

**提示：**
xUnit 的文档可参阅 https://xunit.github.io/。

Visual Studio 测试环境支持其他测试框架。测试适配器，如 NUnit、xUnit、Boost(用于 C++)、Chutzpah(用于 JavaScript)和 Jasmine(用于 JavaScript)可通过扩展和更新来使用；这些测试适配器与 Visual Studio Test Explorer 集成。

xUnit 是.NET Core 中一个杰出的测试框架，也由微软的.NET Core 和 ASP.NET Core 开源代码使用，所以 xUnit 是本节的重点。

## 28.3.1 使用 xUnit 和.NET Core

使用.NET Core 应用程序，可以创建 xUnit 测试，其方式与 MSTest 测试类似。从命令行，可以使用：

```
> dotnet new xunit
```

创建 xUnit 测试项目。在 Visual Studio 2017 中，可以选择项目类型 xUnit Test Project(.NET Core)。

在示例项目中，测试与以前相同的.NET 标准库 UnitTestingSamples。这个库包含之前所示的测试的类型：DeepThought 和 StringSample。测试项目的名称是 UnitTestingSamples.xUnit.Tests。

这个项目需要引用 xunit(对于单元测试，是 xunit.runner.visualstudio[在 Visual Studio 中运行测试])和 UnitTestingSamples 项目(应测试的代码)。为了与.NET Core 命令行集成，添加 dotnet-xunit 的 DotNetCliToolReference(项目文件 UnitTestingSamples.xUnit.Tests/UnitTestingSamples.xUnit.Tests.csproj)。

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
    <IsPackable>false</IsPackable>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.NET.Test.Sdk" Version="15.5.0" />
    <PackageReference Include="xunit" Version="2.3.1" />
    <PackageReference Include="xunit.runner.visualstudio" Version="2.3.1" />
    <DotNetCliToolReference Include="dotnet-xunit" Version="2.3.1" />
  </ItemGroup>

  <ItemGroup>
    <ProjectReference Include="..\UnitTestingSamples\UnitTestingSamples.csproj" />
  </ItemGroup>

</Project>
```

## 28.3.2 创建 Fact 属性

创建测试的方式非常类似于之前的方法。在 MSTest 中，需要给测试类添加属性。但在 xUnit 中是不必要的，因为会在所有的公共类中搜索测试方法。在 MSTest 和 xUnit 中测试方法 TheAnswerToTheUltimateQuestionOfLifeTheUniverseAndEverything 的差异只是带注释和 Fact 特性的测试方法和不同的 Assert.Equal 方法(代码文件 UnitTestingSamples.xUnit.Tests/DeepThoughtTests.cs)：

```
public class DeepThoughtTest
{
  [Fact]
  public void
    ResultOfTheAnswerToTheUltimateQuestionOfLifeTheUniverseAndEverythingTest()
  {
    int expected = 42;
    var dt = new DeepThought();
    int actual =
    dt.TheAnswerToTheUltimateQuestionOfLifeTheUniverseAndEverything();
    Assert.Equal(expected, actual);
  }
}
```

现在使用的 Assert 类在 XUnit 名称空间中定义。与 MSTest 的 Assert 方法相比，这个类定义了更多的方法，用于验证。例如，不是添加一个特性来指定预期的异常，而是使用 Assert.Throws 方法，允许在一个测试方法中多次检查异常(代码文件 UnitTestingSamples.xUnit.Tests/StringSampleTest.cs)：

```
[Fact]
public void GetStringDemoExceptions()
{
  var sample = new StringSample(string.Empty);
  Assert.Throws<ArgumentNullException>(() => sample.GetStringDemo(null, "a"));
  Assert.Throws<ArgumentNullException>(() => sample.GetStringDemo("a", null));
  Assert.Throws<ArgumentException>(() =>
    sample.GetStringDemo(string.Empty, "a"));
}
```

## 28.3.3　创建 Theory 特性

xUnit 为不需要参数的测试方法定义 Fact 特性。使用 xUnit 还可以调用需要参数的单元测试方法；使用 Theory 特性提供数据，添加一个派生于 Data 的特性。这样就可以通过一个方法定义多个单元测试了。

在下面的代码片段中，Theory 特性应用于 GetStringDemoInlineData 单元测试方法。StringSample.GetStringDemo 方法定义了取决于输入数据的不同路径。如果第二个参数传递的字符串不包含在第一个参数中，就到达第一条路径。如果第二个字符串包含在第一个字符串的前 5 个字符中，就到达第二条路径。第三条路径是用 else 子句到达的。要到达所有不同的路径，3 个 InlineData 特性要应用于测试方法。每个特性都定义了 4 个参数，它们以相同的顺序直接发送到单元测试方法的调用中。特性还定义了被测试方法应该返回的值(代码文件 UnitTestingSamples.xUnit.Tests /StringSampleTest.cs)：

```
[Theory]
[InlineData("", "longer string", "nger",
  "removed nger from longer string: lo string")]
[InlineData("init", "longer string", "string", "INIT")]
public void GetStringDemoInlineData(string init, string a, string b,
  string expected)
{
  var sample = new StringSample(init);
  string actual = sample.GetStringDemo(a, b);
  Assert.Equal(expected, actual);
}
```

特性 InlineData 派生于 Data 特性。除了通过特性直接把值提供给测试方法之外，值也可以来自于属性、方法或类。以下例子定义了一个静态方法，它用 IEnumerable<object>对象返回相同的值(代码文件 UnitTestingSamples.xUnit.Tests/StringSampleTest.cs)：

```
public static IEnumerable<object[]> GetStringSampleData() =>
  new[]
  {
    new object[] { "", "a", "b", "b not found in a" },
    new object[] { "", "longer string", "nger",
    "removed nger from longer string: lo string" },
    new object[] { "init", "longer string", "string", "INIT" }
};
```

单元测试方法现在用 MemberData 特性改变了。这个特性允许使用返回 IEnumerable<object>的静态属性或方法，填写单元测试方法的参数(代码文件 UnitTestingSamples.xUnit.Tests/StringSampleTest.cs)：

```
[Theory]
[MemberData("GetStringSampleData")]
public void GetStringDemoMemberData(string init, string a, string b,
  string expected)
{
  var sample = new StringSample(init);
  string actual = sample.GetStringDemo(a, b);
  Assert.Equal(expected, actual);
}
```

### 28.3.4 使用 Mocking 库

下面是一个更复杂的例子：在 MVVM 应用程序中，为客户端服务库创建一个单元测试。本章的示例代码仅包含该应用程序使用的一个库。这个服务使用依赖注入功能，注入接口 IBooksRepository 定义的存储库。用于测试 AddOrUpdateBookAsync 方法的单元测试不应该测试该库，而只测试方法中的功能。对于库，应执行另一个单元测试：下面的代码段显示了类的实现(代码文件 MockingSamples/BooksLib/Services/BooksService.cs)：

```
public class BooksService: IBooksService
{
  private ObservableCollection<Book> _books = new ObservableCollection<Book>();
  private IBooksRepository _booksRepository;
  public BooksService(IBooksRepository repository)
  {
    _booksRepository = repository;
  }

  public async Task LoadBooksAsync()
  {
    if (_books.Count > 0) return;
    IEnumerable<Book> books = await _booksRepository.GetItemsAsync();
    _books.Clear();
    foreach (var b in books)
    {
      _books.Add(b);
    }
  }

  public Book GetBook(int bookId) =>
    _books.Where(b => b.BookId == bookId).SingleOrDefault();

  public async Task<Book> AddOrUpdateBookAsync(Book book)
  {
    if (book == null) throw new ArgumentNullException(nameof(book));

    Book updated = null;
    if (book.BookId == 0)
    {
      updated = await _booksRepository.AddAsync(book);
      if (updated == null) throw new InvalidOperationException();

      _books.Add(updated);
    }
    else
    {
      updated = await _booksRepository.UpdateAsync(book);
      if (updated == null) throw new InvalidOperationException();

      Book old = _books.Where(b => b.BookId == updated.BookId).Single();
      int ix = _books.IndexOf(old);
      _books.RemoveAt(ix);
      _books.Insert(ix, updated);
    }
    return updated;
  }

  public IEnumerable<Book> Books => _books;
}
```

因为 AddOrUpdateBookAsync 的单元测试不应该测试用于 IBooksRepository 的存储库，所以需要实现一个用于测试的存储库。为了简单起见，可以使用一个模拟库自动填充空白。一个常用的模拟库是 Moq。对于单元测试项目，添加 NuGet 包 Moq。

**注意：**
除了使用 Moq 框架之外，还可以用示例数据实现一个内存中的存储库。在用户界面的设计过程中，可以这么做来处理应用程序的示例数据。

使用 xUnit 时，每次运行测试都会创建测试类的一个新实例。如果多个测试需要相同的功能，就可以把这个功能移动到构造函数中。如果每次运行测试后需要释放资源，就可以实现 IDisposable 接口。

在 BooksServiceTest 类的构造函数中，实例化一个 Mock 对象，传递泛型参数 IBooksRepository。Mock 构造函数创建接口的实现代码。因为需要从存储库中得到一些非空结果来创建有用的测试，所以 Setup 方法定义可以传递的参数，ReturnsAsync 方法定义了方法存根返回的结果。使用 Mock 类的 Object 属性访问模拟对象，并传递它，以创建 BooksService 类的实例。有了这些设置，就可以实现单元测试(代码文件 MockingSamples/BooksLib.Tests/Services/BooksServiceTest.cs)：

```
public class BooksServiceTest : IDisposable
{
  private const string TestTitle = "Test Title";
  private const string UpdatedTestTitle = "Updated Test Title";
  public const string APublisher = "A Publisher";
  private BooksService _booksService;
  private Book _newBook = new Book
  {
    BookId = 0,
    Title = TestTitle,
    Publisher = APublisher
  };
  private Book _expectedBook = new Book
  {
    BookId = 1,
    Title = TestTitle,
    Publisher = APublisher
  };
  private Book _notInRepositoryBook = new Book
  {
    BookId = 42,
    Title = TestTitle,
    Publisher = APublisher
  };
  private Book _updatedBook = new Book
  {
    BookId = 1,
    Title = UpdatedTestTitle,
    Publisher = APublisher
  };

  public BooksServiceTest()
  {
    var mock = new Mock<IBooksRepository>();
    mock.Setup(repository =>
      repository.AddAsync(_newBook)).ReturnsAsync(_expectedBook);
    mock.Setup(repository =>
      repository.UpdateAsync(_notInRepositoryBook)).ReturnsAsync(null as Book);
    mock.Setup(repository =>
      repository.UpdateAsync(_updatedBook)).ReturnsAsync(_updatedBook);

    _booksService = new BooksService(mock.Object);
  }
  //...
```

**注意：**
IDisposable 接口参见第 17 章。

实现的第一个单元测试 AddOrUpdateBookAsync_ThrowsForNull 证明，如果把 null 传递给 AddOrUpdateBookAsync 方法，就会抛出 ArgumentNullException 异常。该实现代码只需要在构造函数中实例化成员变量_booksService，而不需要模拟设置。这个代码示例还说明，单元测试方法可以实现为返回 Task 的异步方法(代码文件 MockingSamples/BooksLib.Tests/Services/BooksServiceTest.cs)：

```
[Fact]
public async Task AddOrUpdateBookAsync_ThrowsForNull()
{
  // arrange
  Book nullBook = null;
  // act and assert
  await Assert.ThrowsAsync<ArgumentNullException>(() =>
    _booksService.AddOrUpdateBookAsync(nullBook));
}
```

单元测试方法 AddOrUpdateBook_AddedBookReturnsFromRepository 给服务添加了一本新书(变量_newBook)，并期望返回_expectedBook 对象。在 AddOrUpdateBookAsync 方法的实现代码中，调用了 IBooksRepository 的 AddAsync 方法，因此，应用了以前给这个方法定义的模拟设置。这个方法的结果应是，返回的 Book 等于_expectedBook，_expectedBook 也需要添加到 BooksService 的图书集合中(代码文件 MockingSamples/BooksLib.Tests/Services/BooksServiceTest.cs)：

```
[Fact]
public async Task AddOrUpdateBook_AddedBookReturnsFromRepository()
{
  // arrange in constructor
  // act
  Book actualAdded = await _booksService.AddOrUpdateBookAsync(_newBook);

  // assert
  Assert.Equal(_expectedBook, actualAdded);
  Assert.Contains(_expectedBook, _booksService.Books);
}
```

AddOrUpdateBook_UpdateNotExistingBookThrows 单元测试证明，尝试更新服务中不存在的图书，应抛出 InvalidOperationException 异常(代码文件 MockingSamples/BooksLib.Tests/Services/BooksServiceTest.cs)：

```
[Fact]
public async Task AddOrUpdateBook_UpdateNotExistingBookThrows()
{
  // arrange in constructor
  // act and assert
  await Assert.ThrowsAsync<InvalidOperationException>(() =>
    _booksService.AddOrUpdateBookAsync(_notInRepositoryBook));
}
```

更新图书的常见情形用单元测试 AddOrUpdateBook_UpdateBook 来处理。这里需要做额外的准备，中更新前，先把图书添加到服务中(代码文件 MockingSamples/BooksLib.Tests/Services/BooksServiceTest.cs)：

```
[Fact]
public async Task AddOrUpdateBook_UpdateBook()
{
  // arrange
  await _booksService.AddOrUpdateBookAsync(_newBook);

  // act
  Book updatedBook = await _booksService.AddOrUpdateBookAsync(_updatedBook);

  // assert
  Assert.Equal(_updatedBook, updatedBook);
  Assert.Contains(_updatedBook, _booksService.Books);
}
```

当使用 MVVM 模式与基于 XAML 的应用程序，以及使用 MVC 模式和基于 Web 的应用程序时，会降低用户界面的复杂性，减少复杂 UI 测试的需求。然而，仍有一些场景应该用 UI 测试，例如，浏览页面、拖曳元素等。此时应使用 Visual Studio 的 UI 测试功能。

## 28.4 实时单元测试

Visual Studio 2017 的企业版提供了一个非常好的单元测试功能：实时单元测试。最好尽早看到错误，而能看到它们的第一个地方是 Visual Studio 编辑器。从 Test 菜单中，可以启动 Live Unit Testing。打开 Live Unit Testing 后，可以在编辑器中直接看到测试覆盖的代码行以及测试运行成功的代码(参见图 28-4)。

如果在代码编辑器中引入了一些错误，就可以立即看到这个问题——即使不保存文件也能看到。如果编译器在用户编辑时运行成功，那么与刚刚编辑过的方法相关联的单元测试就会运行，可以看到结果(如图 28-5 所示)，并可以做出相应的反应。

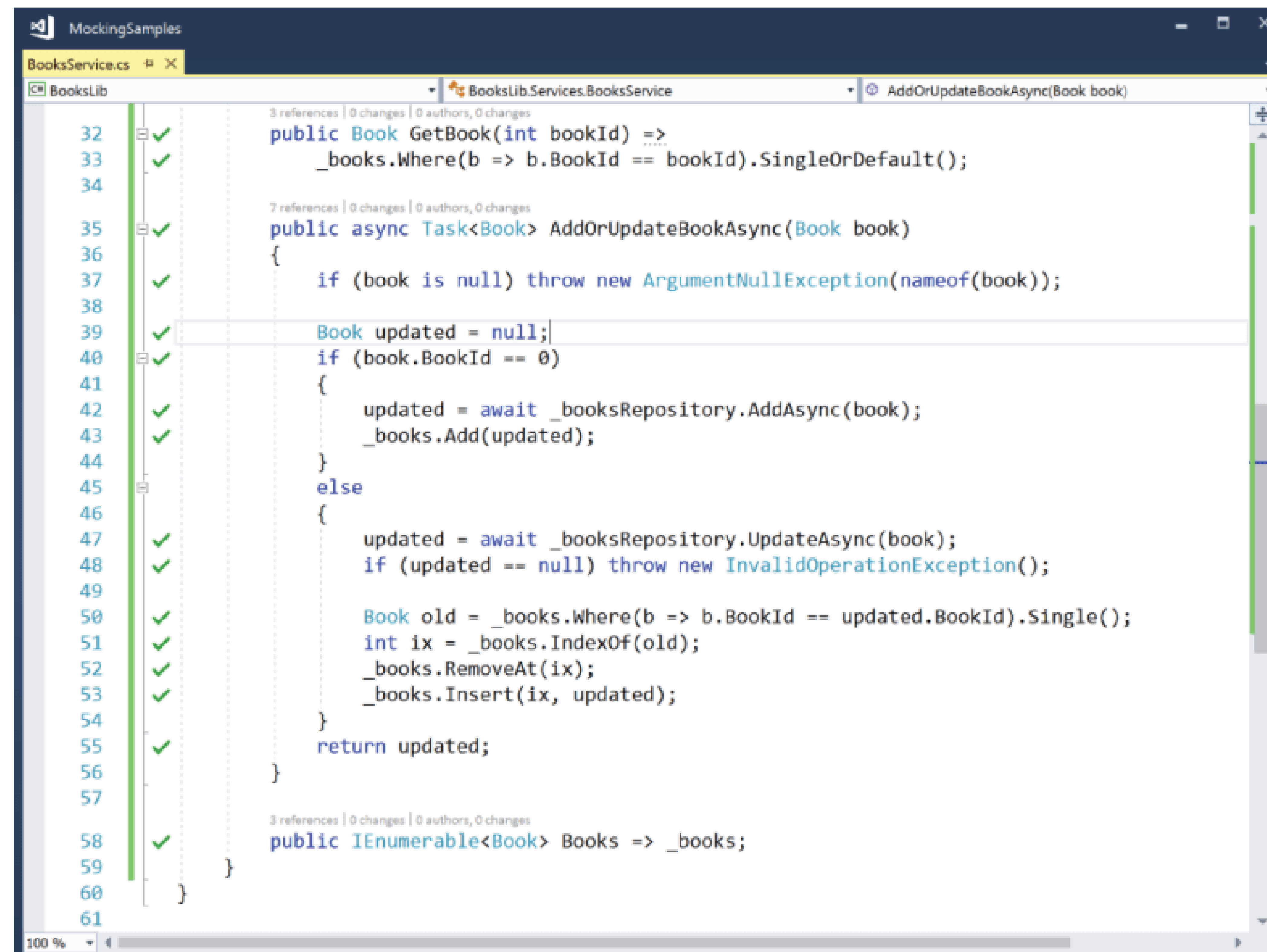


图 28-4

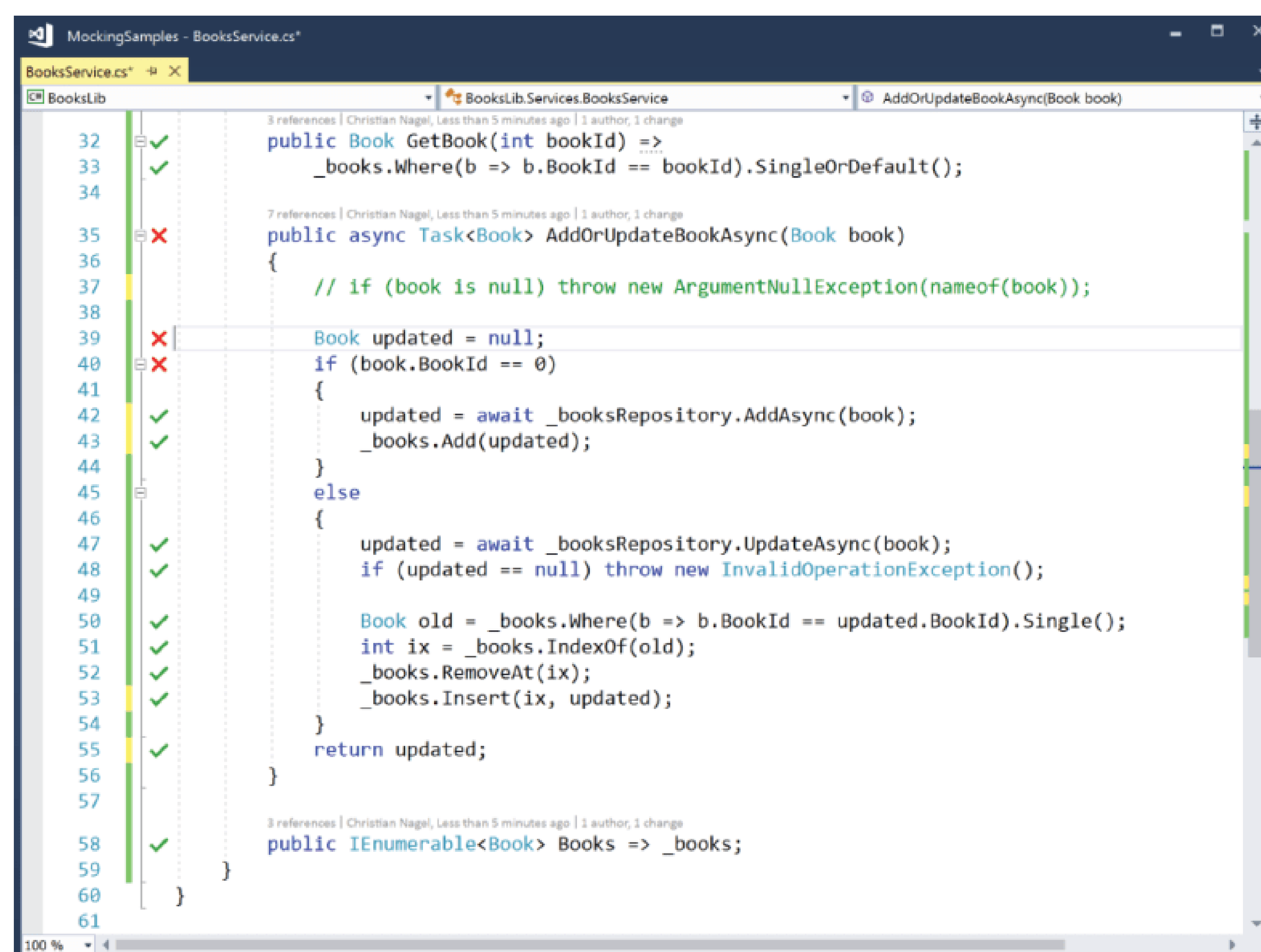


图 28-5

在编辑时运行的所有测试都应该运行得很快。不应该使用 Live Unit Testing 运行集成测试。在 Solution Explorer 中，可以选择要从 Live Unit Testing 中排除或包含的测试项目和测试类。

还可以使用注释排除特定的测试方法——使用 Trait 属性的 xUnit：

```
[Trait("Category", "SkipWhenLiveUnitTesting")]
```

对 MSTest 使用 TestCategory 属性：

```
[TestCategory("SkipWhenLiveUnitTesting")]
```

当电池电量不足 30%时，实时单元测试不能运行。在 Visual Studio 中选择 Tools | Options，可以配置这个设置和其他 Live Unit Test 设置。

## 28.5 使用 EF Core 进行单元测试

创建单元测试时，需要用提供测试数据的测试类替换依赖项。Entity Framework Core(EF Core)上下文中的依赖项又如何呢？通常没有一个接口是由上下文实现的，但是上下文本身(例如，BooksContext)是被注入的。EF Core 基于内存的提供程序提供了一个解决方案，可以将其用作模拟类，而不是使用 EF Core SQL Server 提供程序。

**注意：**
EF Core 详见第 26 章。

下面从一个简单的 Book 类型、BooksContext 和 BooksService 开始。BooksService 类应该在单元测试中进行测试。

Book 是一个简单的类，它保留了一些属性(代码文件 EFCoreSample/EFCoreSample/Book.cs)：

```
public class Book
{
  public int BookId { get; set; }
  public string Title { get; set; }
  public string Publisher { get; set; }
}
```

类 BooksContext 管理到数据库的连接，并将 Book 类型映射到 Books 表(代码文件 EFCoreSample/EFCoreSample/BooksContext.cs)：

```
public class BooksContext : DbContext
{
  public BooksContext(DbContextOptions<BooksContext> options)
    : base(options) { }

  public DbSet<Book> Books { get; set; }
}
```

最后，类 BooksService 通过依赖注入使用 BooksContext，并定义 GetTopBooksByPublisher 方法。这个方法应该只返回 10 本书(代码文件 EFCoreSample/EFCoreSample/BooksService.cs)：

```
public class BooksService
{
  private readonly BooksContext _booksContext;

  public BooksService(BooksContext booksContext)
  {
    _booksContext = booksContext;
  }

  public IEnumerable<Book> GetTopBooksByPublisher(string publisher)
  {
    if (publisher == null) throw new ArgumentNullException(nameof(publisher));

    return _booksContext.Books
      .Where(b => b.Publisher == publisher)
      .Take(10)
      .ToList();
  }
}
```

对于单元测试，创建一个 xUnit 项目。为了使用 EF Core 内存提供程序，除了添加 EFCoreSample 项目之外，还要添加 NuGet 包 Microsoft.EntityFrameworkCore.InMemory。

现在，可以使用 DbContextOptionsBuilder 来创建内存中数据库的选项。UseInMemoryDatabase 是包 Microsoft.EntityFrameworkCore.InMemory 中的扩展方法，用于添加 EF Core 内存提供程序。在 InitContext 方法中，将创建 1000 个 book 对象，并保存到上下文的对象列表中，以备单元测试使用(代码文件 EFCoreSample/EFCoreSample.Tests/BooksServiceTest.cs)：

```
public class BooksServiceTest : IDisposable
{
  private BooksContext _booksContext;
  private const string PublisherName = "A Publisher";
  public BooksServiceTest()
  {
    InitContext();
  }

  private void InitContext()
  {
    var builder = new DbContextOptionsBuilder<BooksContext>()
      .UseInMemoryDatabase("BooksDB");
    _booksContext = new BooksContext(builder.Options);

    // init with 1000 books
    var books = Enumerable.Range(1, 1000)
      .Select(i =>
        new Book
        {
          BookId = i,
          Title = $"Sample {i}",
          Publisher = PublisherName })
      .ToList();
    _booksContext.Books.AddRange(books);
    _booksContext.SaveChanges();
  }

  //...

  public void Dispose()
  {
    _booksContext?.Dispose();
  }
}
```

**注意：**
Enumerable 类和 Range 方法详见第 12 章。

现在，单元测试方法 GetTopBooksByPublisherCount 在 arrange 部分实例化 BooksService，在 act 部分调用 GetTopBooksByPublisher 方法，最后在 assert 部分检查返回的图书数量(代码文件 EFCoreSample/EFCoreSample.Tests/BooksServiceTest.cs)：

```
[Fact]
public void GetTopBooksByPublisherCount()
{
  // arrange
  var booksService = new BooksService(_booksContext);
  // act
  var topbooks = booksService.GetTopBooksByPublisher(PublisherName);
  // assert
  Assert.Equal(10, topbooks.Count());
}
```

## 28.6 使用 Windows 应用程序进行 UI 测试

为了测试用户界面，Visual Studio 为通用 Windows 应用程序、WPF 应用程序和 Windows Forms 提供了 Coded UI Test Project 模板。当新建项目时，可以在 Test 组中找到用于 WPF 和 Windows Forms 的项目模板。但是，这个模板不适用于 Windows 应用程序。通用 Windows 应用程序的项目模板在 Windows Universal 组中。请注意，Windows 应用程序不支持自动记录。

**注意：**
创建 Windows 应用程序详见第 33～36 章。

本章将为一个简单的 Windows 应用程序创建 UI 测试。这个应用程序是本章可下载文件的一部分，因此可以使用它进行测试。这个应用程序只包含用来输入一些文本的 TextBox 控件、Button 控件和 TextBlock 控件，当用户单击按钮时，TextBlock 控件应该显示输入的文本，如图 28-6 所示。

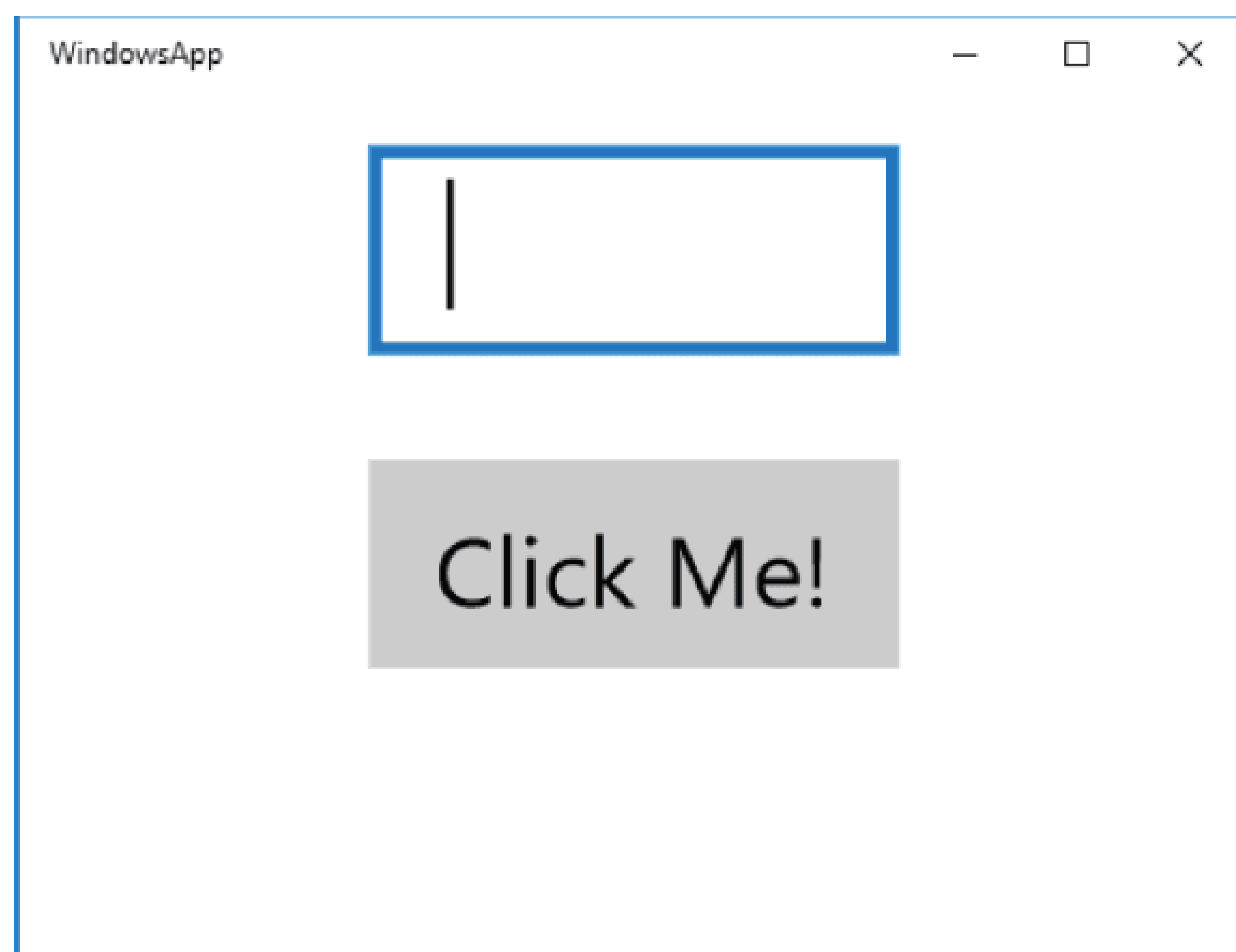



图 28-6

创建新的 Coded UI Test Project(Universal Windows)时，会显示如图 28-7 所示的对话框。选择 Edit UI Map 或 Add Assertions 选项时，可以从正在运行的应用程序中选择控件，该应用程序将控件的自动化对等点添加到映射中，因此可以轻松地以编程方式访问它。

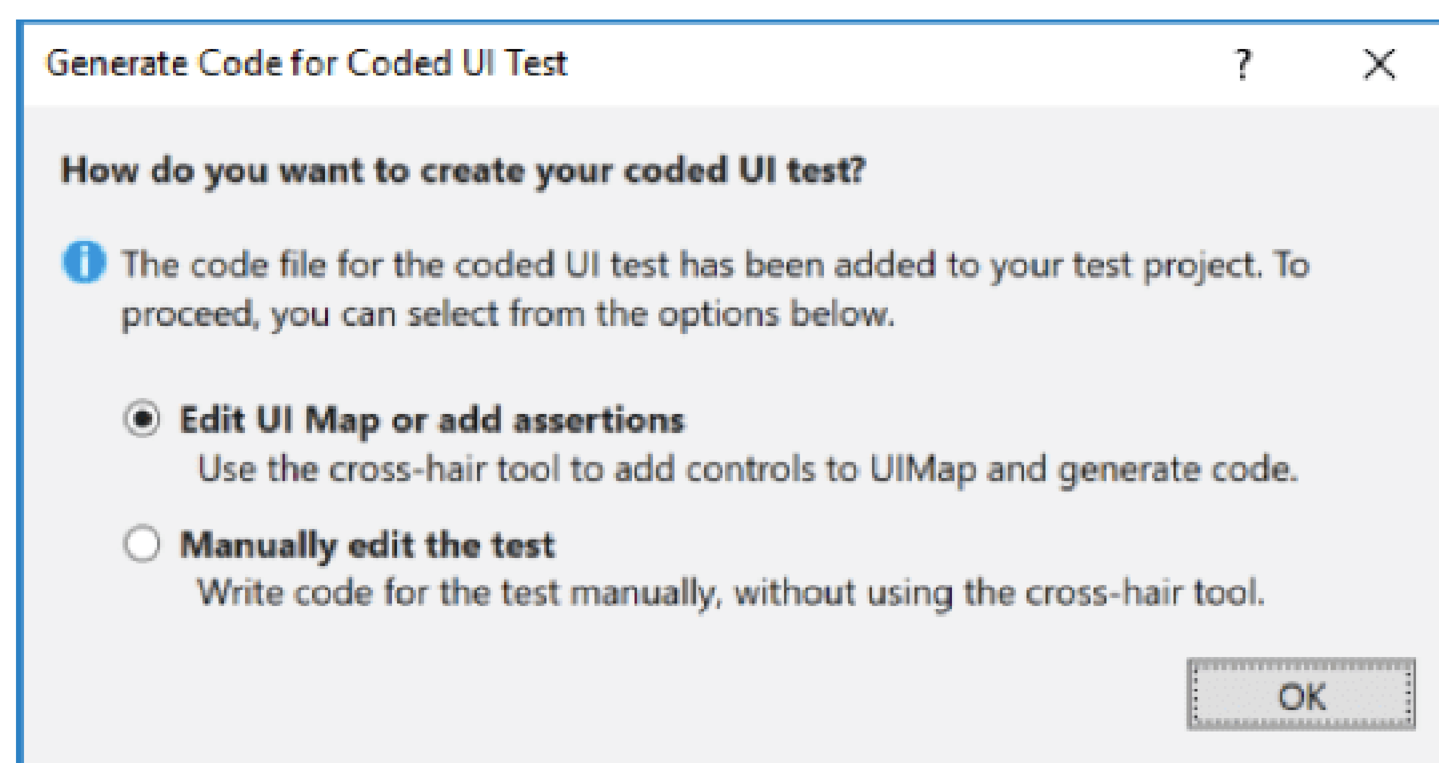



图 28-7

单击 OK 创建新项目后，就打开 Coded UI Test Builder (参见图 28-8)。Windows 应用程序不支持记录操作，所以这个选项是灰色的，但是可以在正在运行的应用程序中拖曳控件上的横线，来向 UI 映射添加控件。

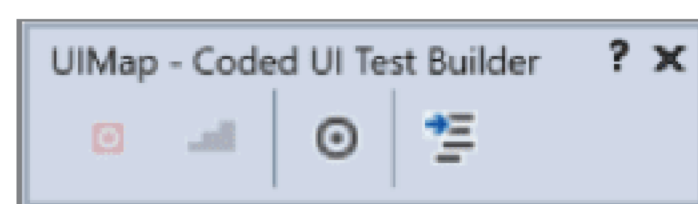



图 28-8

选择控件时，它们的列表显示在工具的左边，而右边部分显示所选控件的属性(参见图 28-9)。需要单击最左边的按钮(Add Control to UI Control Map)，最后将控件添加到映射中。

最后一步，需要在 UIMap - Coded UI Test Builder 窗口中单击 Generate 按钮(见图 28-10)。所显示的对话框包含一个方法，但与 Windows 应用程序一样，仅生成控件，不需要方法名，只需要单击 Generate 按钮。

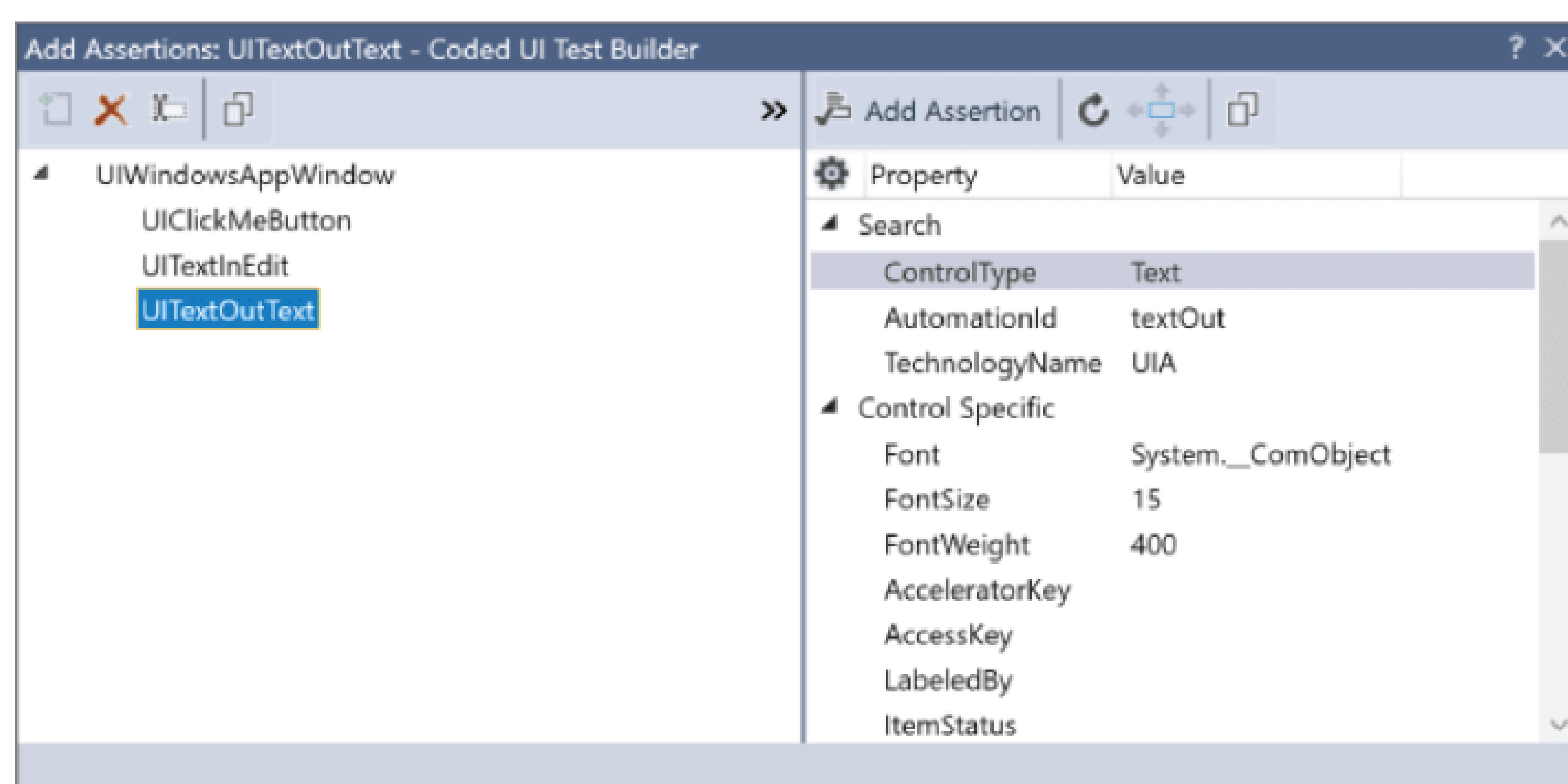



图 28-9

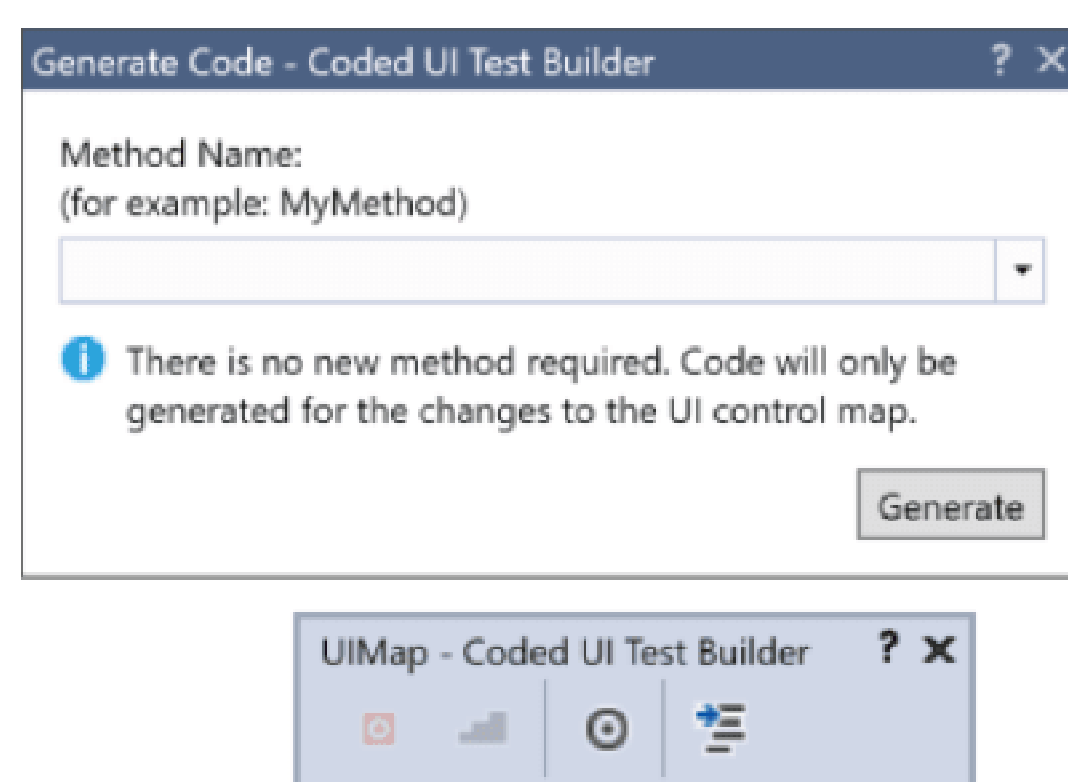



图 28-10

可以在 UIMap.Designer.cs 文件的一个部分类中找到生成的代码。在 UIMap.cs 文件中可以扩展这个类。如果再次启动 Coded UI Test Builder，设计器生成的文件将被覆盖。

设计器生成的代码定义了 UIMap 类，该类包含 UIWindowsAppWindow 的一个属性。这个类表示带有被包含控件的窗口(代码文件 WindowsApp /WindowsApp.UITest/UIMap.Designer.cs)：

```
[GeneratedCode("Coded UITest Builder", "15.0.26621.2")]
public partial class UIMap
{
  public UIWindowsAppWindow UIWindowsAppWindow
  {
    get
    {
      if ((this.mUIWindowsAppWindow == null))
      {
        this.mUIWindowsAppWindow = new UIWindowsAppWindow();
      }
      return this.mUIWindowsAppWindow;
    }
  }
  private UIWindowsAppWindow mUIWindowsAppWindow;
}
```

UIWindowsAppWindow 类派生自基类 XamlWindow，并定义了应用程序启动时使用的 SearchProperties。如果应用程序已经在运行，就可以使用窗口的名称找到应用程序。在示例应用程序中，它设置为 WindowsApp，类名是 Windows.UI.Core.CoreWindow (代码文件 WindowsApp/WindowsApp.UITest/UIMap.Designer.cs)：

```
[GeneratedCode("Coded UITest Builder", "15.0.26621.2")]
public class UIWindowsAppWindow : XamlWindow
{
  public UIWindowsAppWindow()
  {
    this.SearchProperties[XamlControl.PropertyNames.Name] = "WindowsApp";
    this.SearchProperties[XamlControl.PropertyNames.ClassName] =
      "Windows.UI.Core.CoreWindow";
    this.WindowTitles.Add("WindowsApp");
```

```
  }
  //...
```

对于使用 Coded UI Test Builder 选择的每个控件，都会创建一个字段和一个属性。在下面的代码片段中，可以看到为 TextBox 控件生成的代码。每个 UWP 控件都有一个自动化对等点。对于 TextBox 控件，它是 XamlEdit 类。要将 UITextInEdit 属性映射到相应的 TextBox 控件，可以使用 AutomationId。使用附加的属性 AutomationProperties.AutomationId 可以在控件定义中设置 AutomationId。Windows 应用程序的示例代码在文件 MainPage.xaml 的 XAML 代码中定义了这个附加属性。如果没有直接设置 AutomationId，则使用与控件的 Name 属性相同的名称生成它。该属性的 get 访问器在第一次访问属性时实现 XamlEdit 控件的创建。在初始创建之后，返回字段的值(代码文件 WindowsApp/WindowsApp.UITest/UIMap.Designer.cs)：

```
public XamlEdit UITextInEdit
{
  get
  {
    if ((this.mUITextInEdit == null))
    {
      this.mUITextInEdit = new XamlEdit(this);
      this.mUITextInEdit
        .SearchProperties[XamlEdit.PropertyNames.AutomationId] = "textIn";
      this.mUITextInEdit.WindowTitles.Add("WindowsApp");
    }
    return this.mUITextInEdit;
  }
}
private XamlEdit mUITextInEdit;
```

**注意：**
附加属性参见第 33 章。

测试类 MainPageTest 用 CodedUITest 特性进行了注释。这个特性派生自基类 TestClassExtensionAttribute，并识别 UI 测试的类型。要启动应用程序，需要应用程序的自动化 ID。要获得这个 ID，可以使用 Coded UI Test Builder 中的十字，并选择应用程序的磁贴，或者打开 Package.appxmanifest 编辑器中的 Packaging 标签。在 Packaging 选项卡(参见图 28-11)中，可以复制包系列名称，并添加!App 作为后缀。要访问生成的 UIMap，测试类应定义一个属性(代码文件 WindowsApp/WindowsApp.UITest MainPageTest.cs)：

```
[CodedUITest(CodedUITestType.WindowsStore)]
public class MainPageTest
{
  private string TileAutomationId =
    "0e07ecab-af0f-4129-965b-eed7a5beef75_p2wxv0ry6mv8g!App";

  //...

  public TestContext TestContext
  {
    get => testContextInstance;
    set => testContextInstance = value;
  }
  private TestContext testContextInstance;

  public UIMap UIMap
  {
    get
    {
      if (this.map == null)
      {
        this.map = new UIMap();
      }
      return this.map;
    }
  }
  private UIMap map;
}
```

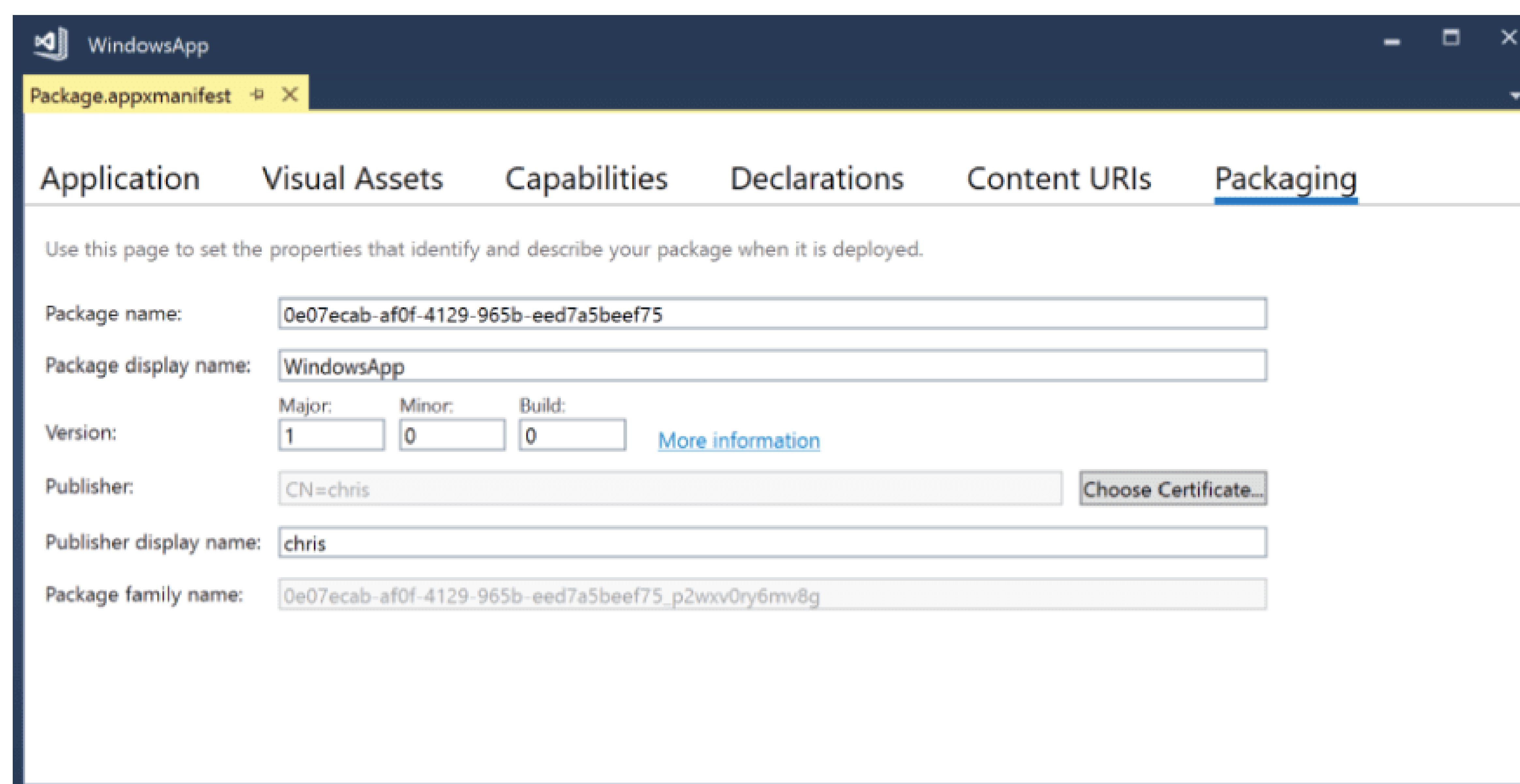



图 28-11

测试在测试方法 EnterTextAndButtonClick 中定义。首先，使用应用程序先前定义的自动化 ID 启动应用程序。接下来，通过设置自动化对等点 XamlEdit 控件的 Text 属性，来填充 TextBox 控件的 Text 属性。诸如按钮单击之类的手势是通过 Gesture 类完成的。Gesture.Tap 轻击或单击控件；在代码片段中，使用 UIClickMeButton(它是关联的 XamlButton 对等控件)选择单击的控件。最后，从与 XamlText 控件关联的 TextBlock 中读取文本(代码文件 WindowsApp/WindowsApp.UITest/MainPageTest.cs)：

```
[TestMethod]
public void EnterTextAndButtonClick()
{
  string inText = "Hello, Windows!";
  XamlWindow xamlWindow = XamlWindow.Launch(TileAutomationId);
  UIMap.UIWindowsAppWindow.UITextInEdit.Text = inText;
  Gesture.Tap(UIMap.UIWindowsAppWindow.UIClickMeButton);
  string outText = UIMap.UIWindowsAppWindow.UITextOutText.DisplayText;
  xamlWindow.Close();
  Assert.AreEqual(inText, outText);
}
```

现在，可以以运行单元测试的方式运行 UI 测试，如本章前面所示。

## 28.7 Web 集成、负载和性能测试

要测试 Web 应用程序，可以创建单元测试，调用控制器、存储库和实用工具类的方法。Tag 辅助程序是简单的方法，在其中，测试可以由单元测试覆盖。单元测试用于测试方法中算法的功能，换句话说，就是方法内部的逻辑。在 Web 应用程序中，创建性能和负载测试也是一个很好的实践。应用程序会伸缩吗？应用程序用一个服务器可以支持多少用户？需要多少台服务器支持特定数量的用户？不容易伸缩的瓶颈是什么？为了回答这些问题，Web 测试可以提供帮助。

ASP.NET Core 提供了一个托管类来创建集成测试。Visual Studio Enterprise 2017 为 Web 负载和性能测试提供了一个记录器，来记录 HTTP 请求。记录器需要在 Internet Explorer 中添加一个插件。

### 28.7.1 ASP.NET Core 集成测试

通过集成测试，可以测试 Web 应用程序从 HTTP 请求到后端的所有内容。应该有比集成测试更多的单元测试。单元测试可能有数千个，但只有几个集成测试。如果单元或集成测试可以覆盖相同的功能，就应该选择单元测试。Visual Studio 的特性“实时单元测试”对于运行单元测试是有用的，但是不应该在集成测试中使用它。需要进行集成测试以获得完整的了解，查看从前端到后端的所有操作。

创建一个 ASP.NET Core 集成测试，使用空模板创建一个 ASP.NET Core Web 应用程序，命名为 ASPNETCoreSample。从生成的代码中运行应用程序，返回字符串“Hello World!”，这将使用 xUnit 进行集成测试。

**注意：**
ASP.NET Core 详见第 30～32 章，以及网上附加第 3 章。

xUnit 项目 ASPNETCoreSample.IntegrationTest 需要一个对 Microsoft.AspNetCore.TestHost 包的引用。这个包包含 TestServer 类来托管和启动 Web 应用程序，并发送请求。还需要对 Web 项目 ASPNETCoreSample 进行引用。

集成测试的常见安排是在构造函数中处理。在这里，实例化 TestServer，将 IWebHostBuilder 传递给 TestServer 类的构造函数。与 ASP.NET Core Web 应用程序的 Program.cs 文件一样，需要进行类似的设置，启用相同的配置文件和相同的中间件(代码文件 ASPNETCoreSample/ASPNETCoreSample. IntegrationTest/AspNetCoreSample Test.cs)：

```
public class ASPNETCoreSampleTest : IDisposable
{
  private TestServer _testServer;

  public ASPNETCoreSampleTest()
  {
    _testServer = new TestServer(
      WebHost.CreateDefaultBuilder().UseStartup<Startup>());
  }

  public void Dispose() => _testServer?.Dispose();

  //...
}
```

在集成测试中，可以使用_testServer 变量创建对主页的请求，并调用返回的 RequestBuilder 的 GetAsync 方法。在使用 Content 属性的 ReadAsStringAsync 读取内容之前，检查返回的 HttpResponseMessage 中成功的状态代码。在 assert 部分，比较结果与预期的字符串(代码文件 ASPNETCoreSample/ASPNETCoreSample.Integration-Test/AspNetCoreSampleTest.cs)：

```
[Fact]
public async Task ReturnHelloWorld()
{
  // act
  RequestBuilder requestBuilder = _testServer.CreateRequest("/");
  HttpResponseMessage response = await requestBuilder.GetAsync();
  response.EnsureSuccessStatusCode();

  var responseString = await response.Content.ReadAsStringAsync();

  // assert
  Assert.Equal("Hello World!", responseString);
}
```

通过 RequestBuilder 类，可以创建 HTTP GET、POST、PUT 等请求，并添加 HTTP 头信息。GetAsync 方法的结果与第 23 章中 HttpClient 类的结果相似。实际上，可以通过调用 CreateClient 方法，从 TestServer 中直接访问 HttpClient 类。CreateWebSocketClient 方法返回一个 WebSocketClient 实例，因此也可以创建 WebSocket 请求。

## 28.7.2 创建 Web 测试

现在已经为 ASP.NET Core 创建了一个集成测试，下面介绍 Visual Studio Enterprise 2017 的一个特性：Web 性能和负载测试。

为了创建 Web 测试，可以选择 Web Application (Model-View-Controller)，确保 Authentication 设置为 Individual User Accounts，创建一个新的 ASP.NET Core Web 应用程序。这个模板内置了足够的功能，允许创建测试。在创建 Web 测试之前，启动应用程序，并向应用程序注册用户。

要创建 Web 测试，需要给解决方案添加一个 Web Performance and Load Test Project，命名为 WebAndLoadTest。单击自动生成的 WebTest1.webtest 文件，打开 Web Test Editor。然后单击 Add Recording 按钮，开始一个 Web 记录。对于这个记录，必须在 Internet Explorer 中安装 Web Test Recorder 插件，该插件随 Visual Studio

一起安装。该记录器记录发送到服务器的所有 HTTP 请求。单击 Web 应用程序 WebApplicationSample 上的一些链接，例如 About 和 Contact，并注册一个新用户。然后单击 Stop 按钮，停止记录。

记录完成后，可以用 Web Test Editor 编辑记录。一个记录如图 28-12 所示。对于所有的请求，可以看到标题信息以及可以影响和改变的表单 POST 数据。

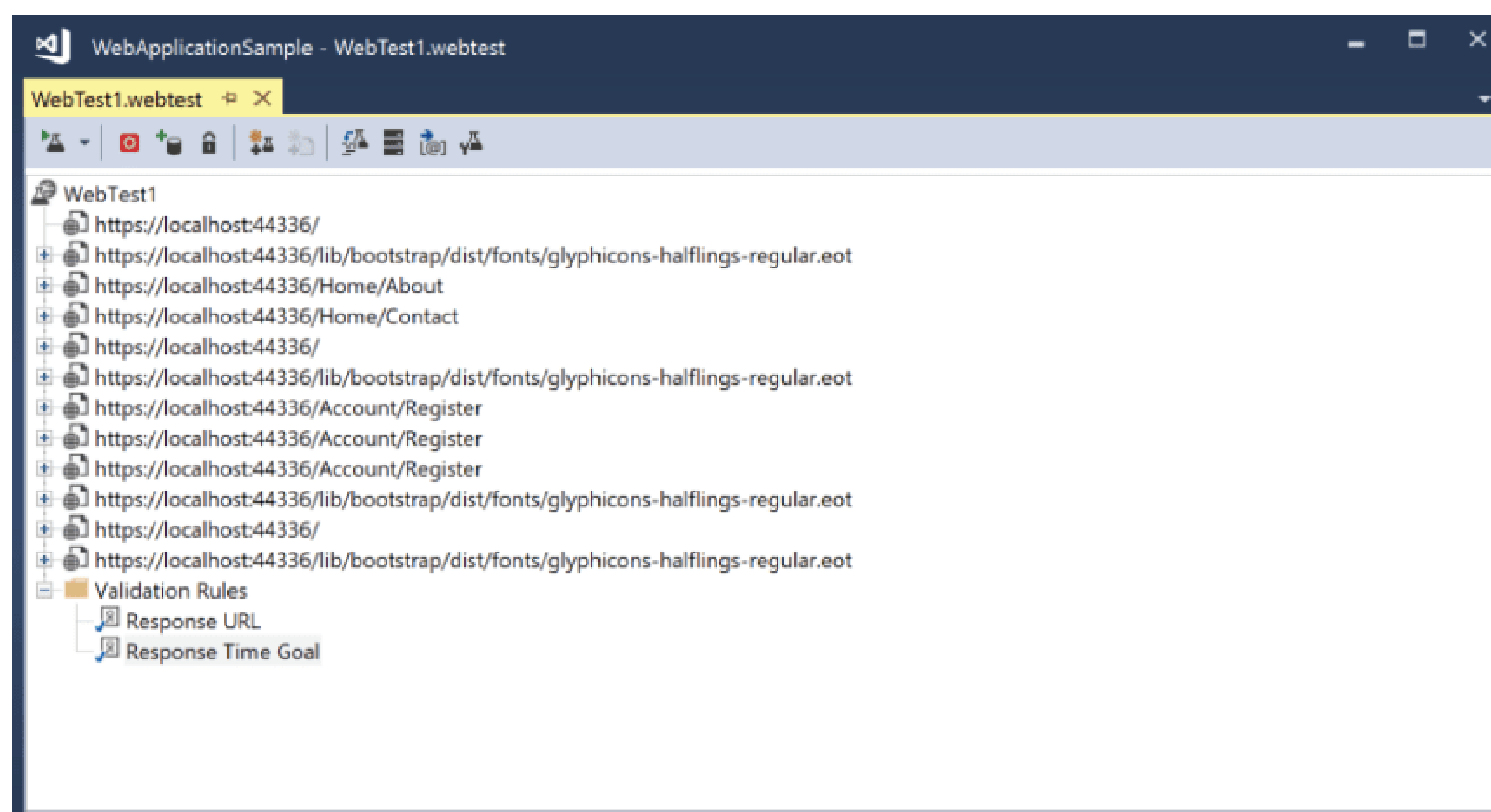



图 28-12

删除不需要的记录，然后单击带有测试名称 NavigateAndRegisterTest 的 Generate Code 按钮，生成源代码，以编程方式发送所有的请求。对于 Web 测试，测试类派生自基类 WebTest，重写了 GetRequestEnumerator 方法。该方法一个接一个地返回请求(代码文件 WebApplicationSample/WebAndLoadTest/NavigateAndRegisterTest.cs)：

```
public class NavigateAndRegisterTest: WebTest
{
  public NavigateAndRegister()
  {
    this.PreAuthenticate = true;
    this.Proxy = "default";
  }

  public override IEnumerator<WebTestRequest> GetRequestEnumerator()
  {
    //...
  }
}
```

方法 GetRequestEnumerator 定义了对网站的请求，例如对 About 页面的请求。对于这个请求，添加一个 HTTP 标题，把该请求定义为源自于主页：

```
public override IEnumerator<WebTestRequest> GetRequestEnumerator()
{
  //...
  WebTestRequest request2 =
    new WebTestRequest("http://localhost:44336/Home/About");
  request2.ThinkTime = 1;
  request2.Headers.Add(new WebTestRequestHeader("Referer",
    "http://localhost:44336/"));
  yield return request2;
  request2 = null;
  //...
}
```

下面发送一个对 Register 页面的 HTTP POST 请求，传递表单数据：

```
WebTestRequest request6 =
  new WebTestRequest("http://localhost:44336/Account/Register");
  request6.Method = "POST";
  request6.ExpectedResponseUrl = "http://localhost:44336/";
  request6.Headers.Add(new WebTestRequestHeader("Referer",
```

```
  "http://localhost:44336/Account/Register"));
 FormPostHttpBody request6Body = new FormPostHttpBody();
 request6Body.FormPostParameters.Add("Email", "sample1@test.com");
 request6Body.FormPostParameters.Add("Password", "Pa$$w0rd");
 request6Body.FormPostParameters.Add("ConfirmPassword", "Pa$$w0rd");
 request6Body.FormPostParameters.Add("__RequestVerificationToken",
 this.Context["$HIDDEN1.__RequestVerificationToken"].ToString());
 request6.Body = request6Body;
 yield return request6;
 request6 = null;
```

在表单中输入一些数据时，最好从数据源中提取数据，以增加灵活性。使用 Web Test Editor，可以添加数据库、CSV 文件或 XML 文件作为数据源(见图 28-13)。使用此对话框可以改变表单参数，从数据源中提取数据。

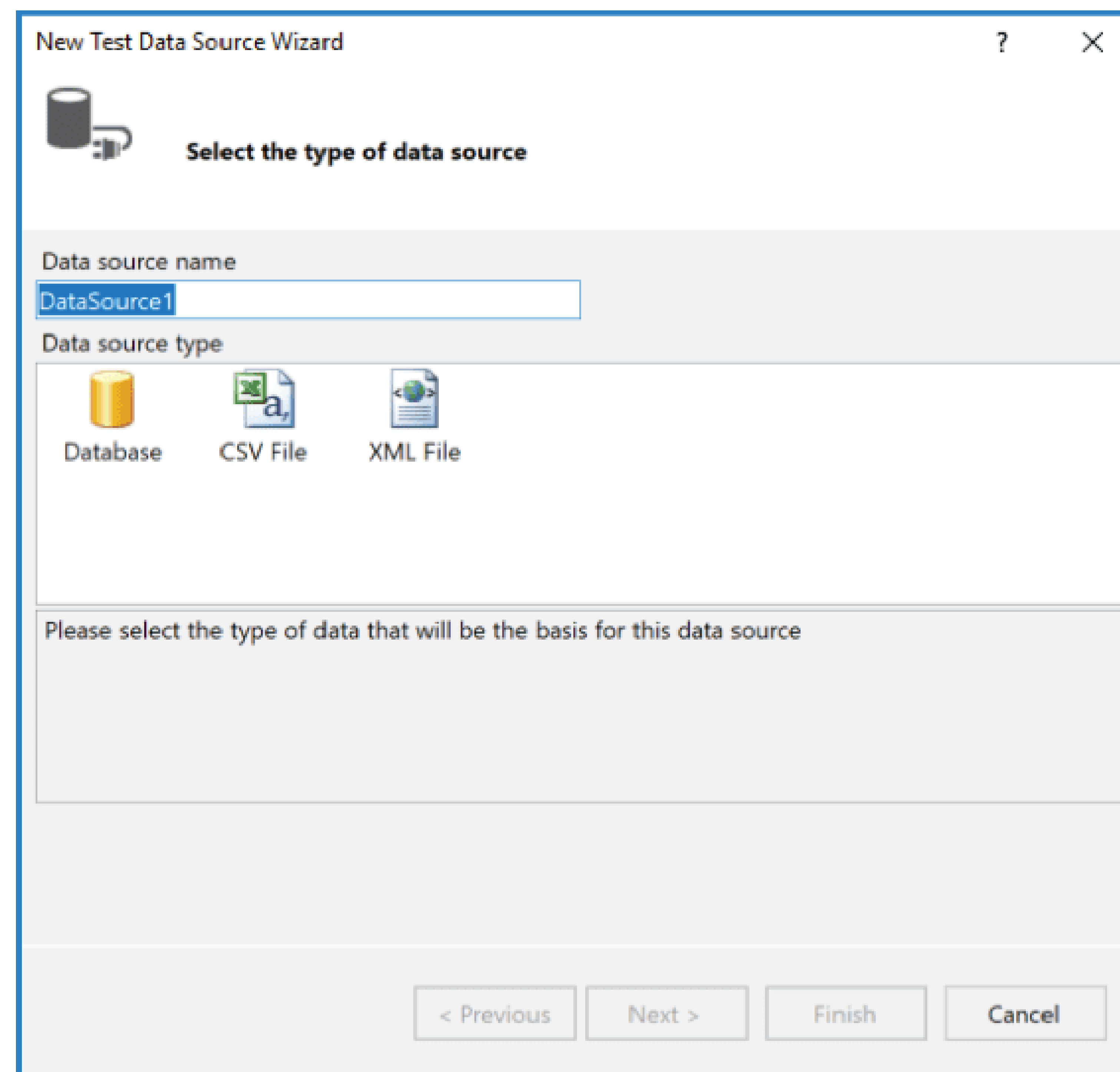



图 28-13

添加数据源就改变了测试代码。对于数据源，测试类用 DeploymentItem 特性(如果使用 CSV 或 XML 文件)、DataSource 和 DataBinding 特性注释：

```
[DeploymentItem("webandloadtestproject\\EmailTests.csv",
 "webandloadtestproject")]
[DataSource("EmailDataSource",
 "Microsoft.VisualStudio.TestTools.DataSource.CSV",
 "|DataDirectory|\\webandloadtestproject\\EmailTests.csv",
 Microsoft.VisualStudio.TestTools.WebTesting.DataBindingAccessMethod
  .Sequential,
 Microsoft.VisualStudio.TestTools.WebTesting.DataBindingSelectColumns
  .SelectOnlyBoundColumns, "EmailTests#csv")]
[DataBinding("EmailDataSource", "EmailTests#csv", "sample1@test#com",
 "EmailDataSource.EmailTests#csv.sample1@test#com")]
public class NavigateAndRegister1: WebTest
{
 //...
}
```

现在，在代码中，可以使用 WebTest 的 Context 属性访问数据源，该属性返回一个 WebTestContext，以通过索引访问所需的数据源：

```
request6Body.FormPostParameters.Add("Email",
this.Context["EmailDataSource.EmailTests#csv.sample1@test#com"].ToString());
```

### 28.7.3 运行 Web 测试

有了测试后，就可以启动测试了。可以直接在 Web Test Editor 中运行并调试测试。在开始测试之前，记得

要启动 Web 应用程序。在 Web Test Editor 中运行测试时，可以看到生成的 Web 页面以及请求和响应的细节信息，如图 28-14 所示。

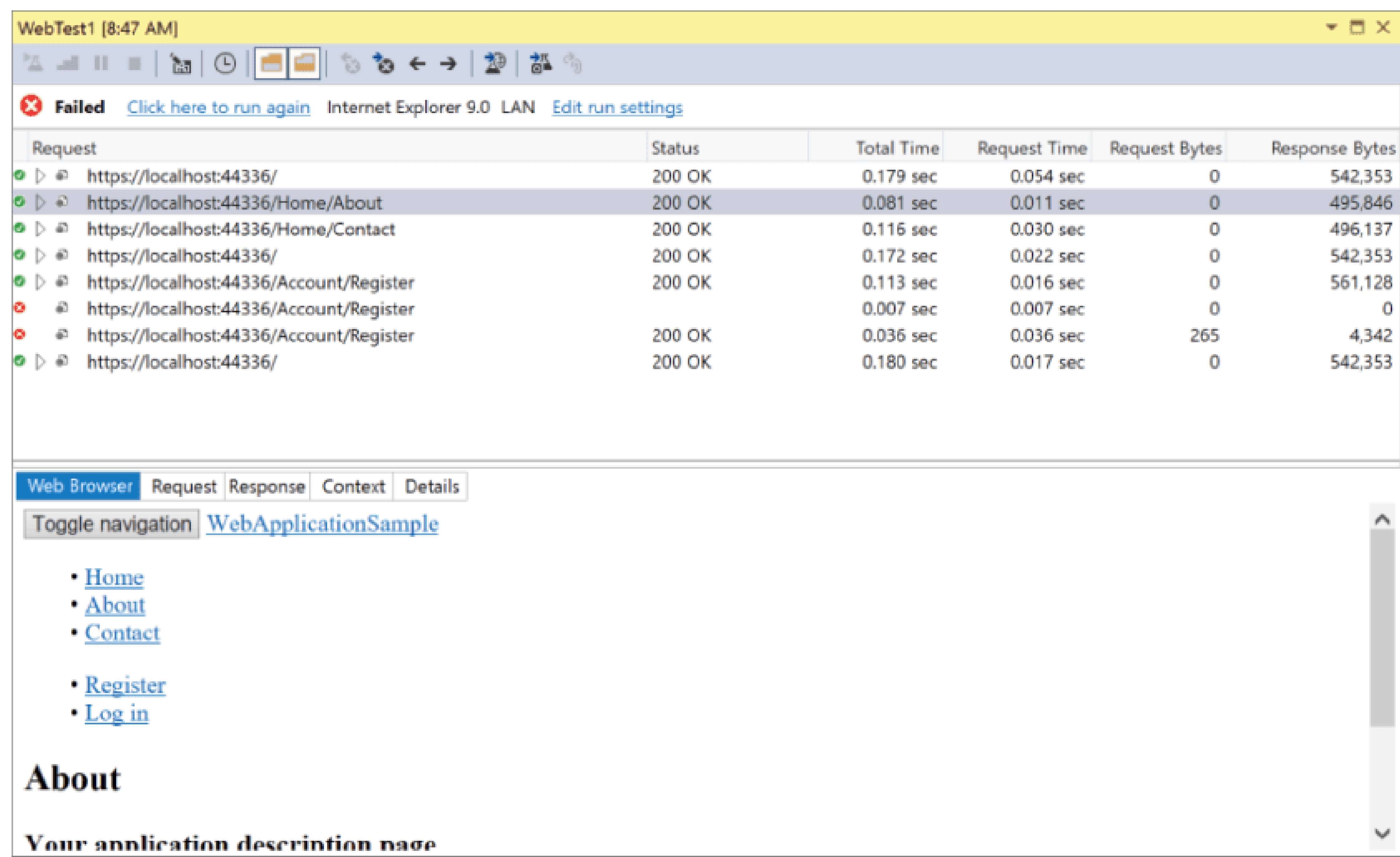


图 28-14

使用 Web Load Tests，可以在 Web 应用程序上模拟高负载。单击 Local.testsettings 解决方案项，可以选择在 Visual Studio Team Services 上运行测试，在多个服务器上运行测试(参见图 28-15)。示例则使用本地机器。

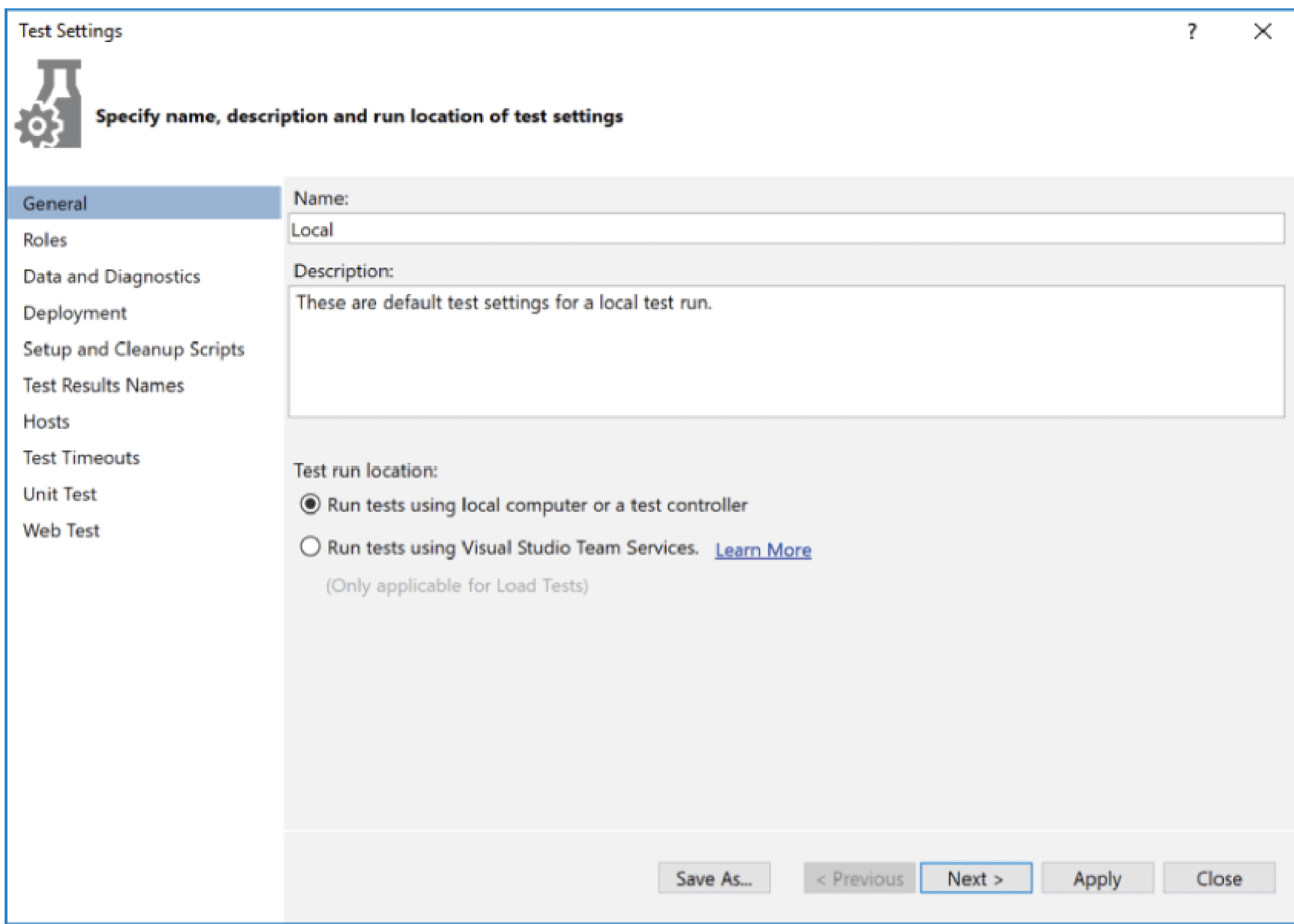


图 28-15

在 Web Test 对话框中，可以通过指定浏览器类型、模拟思考时间和多次运行测试来确定测试运行情况(参见图 28-16)。

**注意：**

使用 Visual Studio 对 Web 应用程序进行负载测试，需要使用 Visual Studio 的企业版。如果没有企业版，可以查看另一种选择：Selenium—— http:// www.seleniumhq.org/。

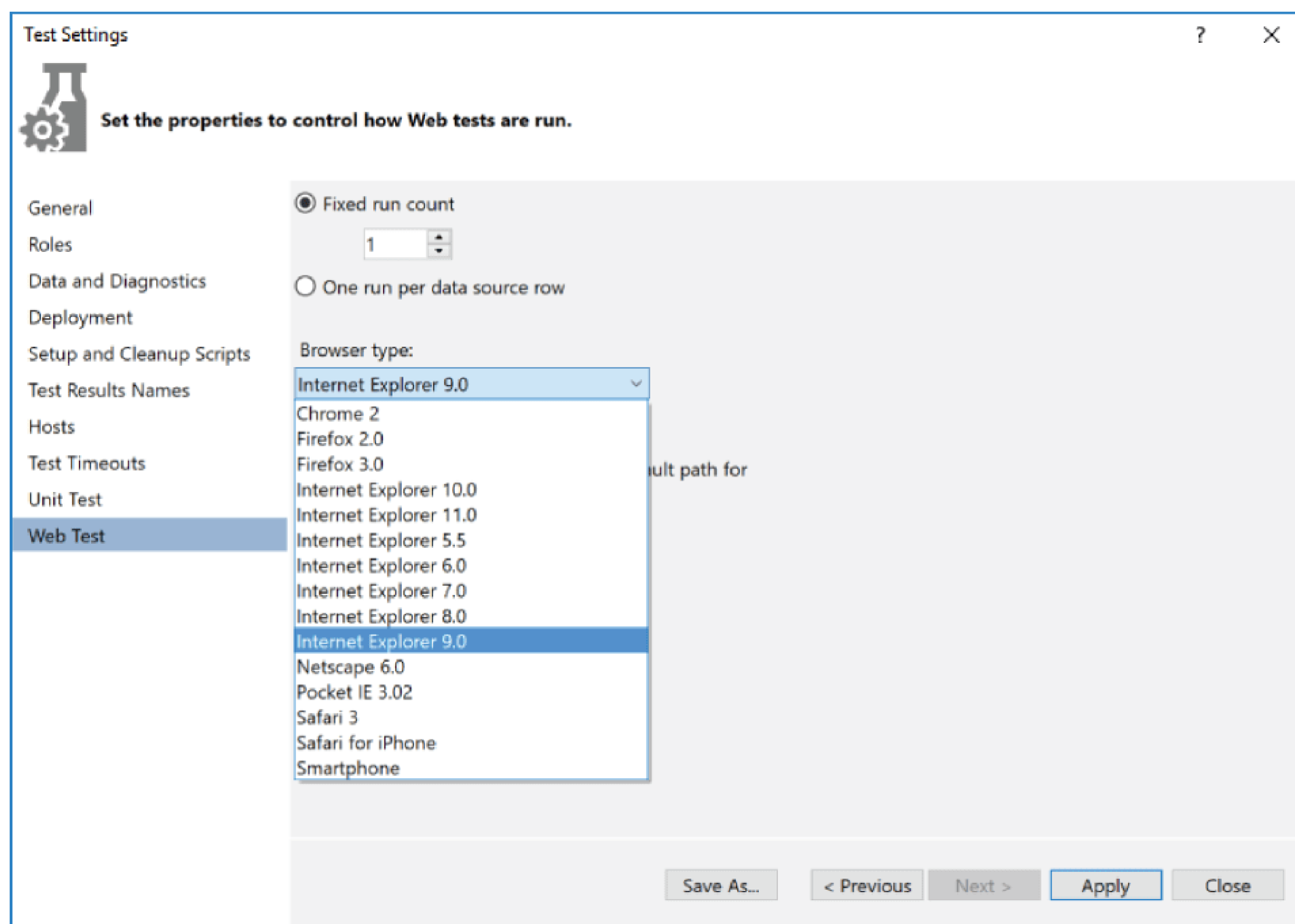



图 28-16

## 28.8 小结

本章介绍了对测试应用程序最重要的方面：创建单元测试、编码的 UI 测试和 Web 测试。

Visual Studio 提供了 Test Explorer 来运行单元测试，而无论它们是用 MSTest 还是 xUnit 创建的。xUnit 的优势是支持.NET Core。本章还介绍了 3A 动作：安排(Arrange)、行动(Act)和断言(Assert)。

在编码的 UI 测试中，学习了如何创建记录，改编记录，根据需要修改所需的 UI 测试代码。

在 ASP.NET Core Web 应用程序中，介绍了如何将 TestHost 类用于集成测试，本章介绍了 Visual Studio 的 Web 负载和性能测试。

测试有助于在部署应用程序之前解决问题，而第 29 章帮助解决正在运行的应用程序的问题。

第 29 章

# 跟踪、日志和分析

**本章要点**

- 用 EventSource 进行简单的跟踪
- 用 EventSource 进行高级跟踪
- 创建自定义跟踪侦听器
- 使用 ILogger 接口
- 给 Windows 应用程序使用 Visual Studio App Center

**本章源代码下载地址(wrox.com)：**

打开 www.wrox.com 的 Download Code 选项卡可下载本章源代码。源代码也可以在 Diagnostics 目录的 https://github.com/ProfessionalCSharp/ProfessionalCSharp7 中找到。本章代码分为以下几个主要的示例文件：

- SimpleEventSourceSample
- EventSourceSampleInheritance
- EventSourceSampleAnnotations
- ClientApp/MyApplicationEvents
- LoggingSample
- WinAppAnalytics

## 29.1 诊断概述

应用程序的发布周期变得越来越短，了解应用程序在生产环境中运行时的行为越来越重要。会发生什么异常？知道使用了什么功能也是要关注的。用户找到应用程序的新功能了吗？他们在页面上停留多长时间?为了回答这些问题，需要应用程序的实时信息。

获得应用程序的信息时，需要区分日志、跟踪和分析。对于日志，错误信息记录到集中的位置上。这些信息由系统管理员用于查找应用程序的问题。跟踪有助于找出哪个方法调用了什么方法。这些信息可用于开发，应用程序在生产环境下运行时，应关闭它。对于.NET，这个技术可通过名称空间 System.Diagnostics 中的类用于日志和跟踪。分析提供了用户的信息：他们在什么地方，使用的操作系统版本是什么，使用了应用程序中的什么功能等。这有助于根据位置、硬件或操作系统，确定应用程序是否有什么问题。它还有助于理解用户当前

的操作。如果很难找到应用程序的某个新功能，用户就可能找不到它。

本章介绍如何获得正在运行的应用程序的实时信息，找出应用程序在生产过程中出现某些问题的原因，或者监视需要的资源，以确保适应较高的用户负载。这就是名称空间 System.Diagnostics.Tracing 的作用。这个名称空间提供了使用 Event Tracing for Windows (ETW)进行跟踪的类。

当然，在应用程序中标记错误的一种方式是抛出异常。然而，有可能应用程序不抛出异常，但仍不像期望的那样运行。应用程序可能在大多数系统上都运行良好，只在几个系统上出问题。在实时系统上，可以启动跟踪收集器，改变日志行为，获得应用程序运行状况的详细实时信息。这可以用 ETW 功能来实现。

如果应用程序出了问题，就需要通知系统管理员。事件查看器是一个常用的工具，并不是只有系统管理员才需要使用它，软件开发人员也需要它。使用事件查看器可以交互地监视应用程序的问题，通过添加订阅功能来了解发生的特定事件。ETW 允许写入应用程序的相关信息。

Application Insights 是一个 Microsoft Azure 云服务，可以监视云中的应用程序。只需要几行代码，就可以得到如何使用应用程序或服务的详细信息。

Visual Studio App Center 允许监控 Windows 和 Xamarin 应用程序。注册了该应用程序后，只需要几行代码就可以接收到应用程序的有用信息。

本章解释了这些功能，演示了如何为应用程序使用它们。

## 29.2 使用 EventSource 跟踪

利用跟踪功能可以从正在运行的应用程序中查看消息。为了获得关于正在运行的应用程序的信息，可以在调试器中启动应用程序。在调试过程中，可以单步执行应用程序，在特定的代码行上设置断点，在满足某些条件时设置断点。调试的问题是包含发布代码的程序与包含调试代码的程序以不同的方式运行。例如，程序在断点处停止运行时，应用程序的其他线程也会挂起。另外，在发布版本中，编译器生成的输出进行了优化，因此会产生不同的效果。在经过优化的发布代码中，垃圾收集要比在调试代码中更加积极。方法内的调用次序可能发生变化，甚至一些方法会被彻底删除，改为就地调用。此时也需要从程序的发布版本中获得运行时信息。跟踪消息要写入调试代码和发布代码中。

下面的场景描述了跟踪功能的作用。在部署应用程序后，它运行在一个系统中时没有问题，而在另一个系统上很快出现了问题。在出问题的系统上打开详细的跟踪功能，就会获得应用程序中所出现问题的详细信息。在运行没有问题的系统上，将跟踪功能配置为把错误消息重定向到 Windows 事件日志系统中。系统管理员会查看重要的错误，跟踪功能的系统开销非常小，因为仅在需要时配置跟踪级别。

.NET 中的跟踪有相当长的历史。.NET 的第一个版本只有简单的跟踪功能和 Trace 类，而.NET 2.0 对跟踪进行了巨大的改进，引入了 TraceSource 类。TraceSource 背后的架构非常灵活，分离出了源代码、侦听器和一个开关，根据一组跟踪级别来打开和关闭跟踪功能。

从.NET 4.5 开始，又引入了一个新的跟踪类 EventSource，并在.NET 4.6 中增强。这个类在 NuGet 包 System.Diagnostics 的 System.Diagnostics.Tracing 名称空间中定义。

新的跟踪架构基于 Windows Vista 中引入的 Event Tracing for Windows(ETW)。它允许在系统范围内快速传递消息，Windows 事件日志记录和性能监视功能也使用它。

下面看看 ETW 跟踪和 EventSource 类的概念。

- ETW 提供程序是一个触发 ETW 事件的库。本章创建的应用程序是 ETW 提供程序。
- ETW 清单描述了可以在 ETW 提供程序中触发的事件。使用预定义清单的优点是，只要安装了应用程序，系统管理员就已经知道应用程序可以触发的事件了。这样，管理员就可以配置特定事件的侦听。新版本的 EventSource 支持自描述的事件和清单描述的事件。
- ETW 关键字可以用来创建事件的类别。它们定义为位标志。
- ETW 任务是分组事件的另一种方式。任务可以基于程序的不同场景来创建，以定义事件。任务通常和操作码一起使用。

- ETW 操作码识别任务中的操作。任务和操作码都用整型值定义。
- 事件源是触发事件的类。可以直接使用 EventSource 类，或创建一个派生自基类 EventSource 的类。
- 事件方法是事件源中触发事件的方法。派生自 EventSource 类的每个 void 方法，如果没有用 NonEvent 特性加以标注，就是一个事件方法。事件方法可以使用 Event 特性来标注。
- 事件级别定义了事件的严重性或冗长性。这可以用于区别关键、错误、警告、信息和详细级事件。
- ETW 通道是事件的接收器。事件可以写入通道和日志文件。Admin、Operational、Analytic 和 Debug 是预定义的通道。

使用 EventSource 类时，要运用 ETW 概念。

## 29.2.1　EventSource 的简单用法

使用 EventSource 类的示例代码利用如下名称空间：

System
System.Collections.Generic
System.Diagnostics.Tracing
System.IO
System.Net.Http
System.Threading.Tasks

EventSource 类提供了各种使用方法。有一种简单的方法实例化这个类，调用方法进行日志记录，还有一种更高级的方法把它用作基类。也有一种添加注释的方法。

使用 EventSource 的第一个例子显示了一个在小型项目中使用的简单案例。在 Console App(.NET Core) 项目中，将 EventSource 实例化为 Program 类的一个静态成员。在构造函数中，指定了事件源的名称(代码文件 SimpleEventSourceSample /Program.cs)：

```
private static EventSource sampleEventSource =
  new EventSource("Wrox-EventSourceSample1");
```

在 Program 类的 Main()方法中，事件源的唯一标识符使用 Guid 属性检索。这个标识符基于事件源的名称创建。之后，编写第一个事件，调用 EventSource 的 Write 方法。所需的参数是需要传递的事件名。其他参数可通过对象的重载使用。第二个传递的参数是定义 Info 属性的匿名对象。它可以把关于事件的任何信息传递给事件日志(代码文件 SimpleEventSourceSample/Program.cs)：

```
static async Task Main()
{
  Console.WriteLine($"Log Guid: {sampleEventSource.Guid}");
  Console.WriteLine($"Name: {sampleEventSource.Name}");
  sampleEventSource.Write("Startup", new { Info = "started app" });
  await NetworkRequestSampleAsync();
  Console.ReadLine();
  sampleEventSource?.Dispose();
}
```

**注意：**
*不是把带有自定义数据的匿名对象传递给 Write 方法，而是可以创建一个类，它派生自基类 EventSource，用 EventData 特性标记它。这个特性在本章后面介绍。*

在 Main()方法中调用的 NetworkRequestSampleAsync()方法发出一个网络请求，写入一个跟踪日志，把请求的 URL 发送到跟踪信息中。完成网络调用后，再次写入跟踪信息。异常处理代码显示了写入跟踪信息的另一个方法重载。不同的重载版本允许传递下一节介绍的特定信息。下面的代码片段显示了设置跟踪级别的 EventSourceOptions。写入错误信息时设定 Error 事件级别。这个级别可以用来过滤特定的跟踪信息。在过滤时，可以决定是只读取错误信息(例如，错误级别信息和比错误级别更重要的信息)。在另一个跟踪会话期间，可以决定使用详细级别读取所有的跟踪信息。EventLevel 枚举定义的值有 LogAlways、Critical、Error、Warning、

Informational 和 Verbose(代码文件 SimpleEventSourceSample/Program.cs)：

```
private static async Task NetworkRequestSample()
{
  try
  {
    using (var client = new HttpClient())
    {
      string url = "http://www.cninnovation.com";
      sampleEventSource.Write("Network", new { Info = $"requesting {url}" });
      string result = await client.GetStringAsync(url);
      sampleEventSource.Write("Network",
        new
        {
          Info =
            $"completed call to {url}, result string length: {result.Length}"
        });
    }
    Console.WriteLine("Complete.................");
  }
  catch (Exception ex)
  {
    sampleEventSource.Write("Network Error",
      new EventSourceOptions { Level = EventLevel.Error },
      new { Message = ex.Message, Result = ex.HResult });
    Console.WriteLine(ex.Message);
  }
}}
```

在运行应用程序之前，需要进行一些准备工作：下载并配置用于读取跟踪信息的工具。下一节将解释如何这样做。

## 29.2.2 跟踪工具

为了分析跟踪信息，可以使用几种工具。logman 工具是 Windows 的一部分。使用 logman，可以创建和管理事件跟踪会话，把 ETW 跟踪信息写入二进制日志文件。tracerpt 也可用于 Windows。这个工具允许将从 logman 写入的二进制信息转换为 CSV、XML 或 EVTX 文件格式。PerfView 工具提供了 ETW 跟踪的图形化信息。

### 1. logman

下面开始使用 logman 从以前创建的应用程序中创建一个跟踪会话。需要先启动应用程序，复制为应用程序创建的 GUID。需要这个 GUID 和 logman 启动日志会话。start 选项开始一个新的会话来进行记录。-p 选项定义了提供程序的名称；这里的 GUID 用来确定提供程序。-o 选项定义了输出文件，-ets 选项直接把命令发送给事件跟踪系统，不需要调度。确保在有写入权限的目录中启动 logman，否则它就不能写入输出文件 mytrace.etl：

```
logman start mysession -p {3b0e7fa6-0346-5781-db55-49d84d7103de}
-o mytrace.etl -ets
```

运行应用程序之后，可以用 stop 命令停止跟踪会话：

```
logman stop mysession -ets
```

**注意：**

logman 有更多的命令，这里不做介绍。使用 logman 可以看到所有已安装的 ETW 跟踪提供程序、它们的名字和标识符，创建数据收集器，在指定的时间启动和停止，定义最大日志文件的大小等。使用 logman - h 可以看到 logman 的不同选项。

### 2. tracerpt

日志文件是二进制格式。为了得到可读的表示，可以使用实用工具 tracerpt。有了这个实用工具，指定-of 选项，可以提取 CSV、XML 和 EVTX 格式：

```
tracerpt mytrace.etl -o mytrace.xml -of XML
```

现在，信息可以用可读的格式获得。有了应用程序记录的信息，就可以在 Task 元素中看到传递给 Write 方

法的事件名，也可以找到 EventData 元素内的匿名对象：

```
<Event xmlns="http://schemas.microsoft.com/win/2004/08/events/event">
  <System>
    <Provider Name="Wrox-SimpleEventSourceSample"
      Guid="{3b0e7fa6-0346-5781-db55-49d84d7103de}" />
    <EventID>2</EventID>
    <Version>0</Version>
    <Level>5</Level>
    <Task>0</Task>
    <Opcode>0</Opcode>
    <Keywords>0x0</Keywords>
    <TimeCreated SystemTime="2017-12-23T10:42:07.960330900+00:59" />
    <Correlation ActivityID="{00000000-0000-0000-0000-000000000000}" />
    <Execution ProcessID="12700" ThreadID="19340" ProcessorID="1"
      KernelTime="30" UserTime="15" />
    <Channel />
    <Computer />
  </System>
  <EventData>
    <Data Name="Info">started app</Data>
  </EventData>
  <RenderingInfo Culture="en-US">
    <Task>Startup</Task>
  </RenderingInfo>
</Event>
```

错误信息与跟踪信息一起显示，如下所示：

```
<EventData>
  <Data Name="Message">An error occurred while sending the request.</Data>
  <Data Name="Result">-2146233088</Data>
</EventData>
```

### 3. PerfView

读取跟踪信息的另一个工具是 PerfView。可以从 Microsoft 下载页面(http://www.microsoft.com/downloads/details.aspx?id=28567)上下载这个工具。这个工具的 1.9 版本有很大改进，可将它用于 Visual Studio 2017 和 EventSource 中自描述的 ETW 格式。这个工具不需要安装，只需要把它复制到需要的地方即可。启动这个工具后，它使用它所在的子目录，并允许直接打开二进制 ETL 文件。图 29-1 显示了 PerfView 打开 logman 创建的文件 mytrace.etl。

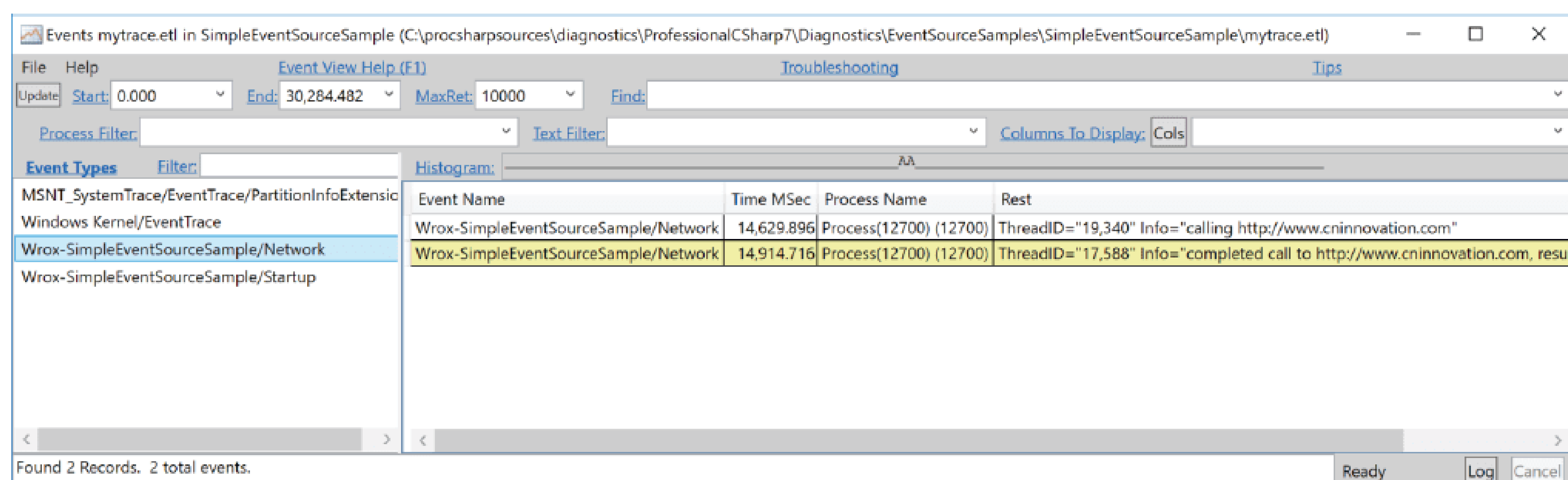



图 29-1

## 29.2.3　派生自 EventSource

除了直接使用 EventSource 的实例之外，最好在一个地方定义所有可以追踪的信息。对于许多应用程序而言，定义一个事件源就足够了。这个事件源可以在一个单独的日志程序集中定义。事件源类需要派生自基类 EventSource。有了这个自定义类，所有应写入的跟踪信息就可以用独立的方法来定义，这些独立方法调用基类的 WriteEvent 方法。类的实现采用单例模式，提供一个静态的 Log 属性，返回一个实例。把这个属性命名为 Log 是使用事件源的一个惯例。私有构造函数调用基类的构造函数，设置事件源名称(代码文件

EventSourceSampleInheritance/SampleEventSource.cs)：

```
public class SampleEventSource : EventSource
{
  private SampleEventSource()
    : base("Wrox-SampleEventSource2") { }

  public static SampleEventSource Log = new SampleEventSource();

  public void Startup() => WriteEvent(1);

  public void CallService(string url) => WriteEvent(2, url);

  public void CalledService(string url, int length) =>
    WriteEvent(3, url, length);


  public void ServiceError(string message, int error) =>
    WriteEvent(4, message, error);
}
```

事件源类的所有 void()方法都用来写入事件信息。如果定义一个辅助方法，就需要用 NonEvent 特性加以标记。

在只应写入信息性消息的简单场景中，不需要其他内容。除了把事件 ID 传递给跟踪日志之外，WriteEvent 方法有 18 个重载版本，允许传递消息 string、int 和 long 值，以及任意数量的 object。

在这个实现代码中，可以使用 SampleEventSource 类型的成员写入跟踪消息，如 Program 类所示。Main()方法使跟踪日志调用 Startup()方法，调用 NetworkRequestSample()方法，通过 CallService()方法创建一个跟踪日志，并使跟踪日志避免错误(代码文件 EventSourceSampleInheritance/Program.cs)：

```
public class Program
{
  Static async Task Main()
  {
    SampleEventSource.Log.Startup();
    Console.WriteLine($"Log Guid: {SampleEventSource.Log.Guid}");
    Console.WriteLine($"Name: {SampleEventSource.Log.Name}");
    await NetworkRequestSampleAsync();
    Console.ReadLine();
  }

  private static async Task NetworkRequestSampleAsync()
  {
    try
    {
      var client = new HttpClient();
      string url = "http://www.cninnovation.com";
      SampleEventSource.Log.CallService(url);
      string result = await client.GetStringAsync(url);
      SampleEventSource.Log.CalledService(url, result.Length);
      Console.WriteLine("Complete.................");
    }
    catch (Exception ex)
    {
      SampleEventSource.Log.ServiceError(ex.Message, ex.HResult);
      Console.WriteLine(ex.Message);
    }
  }
}
```

用这些命令，在项目目录的开发命令提示符下运行应用程序时，会产生一个 XML 文件，其中包含跟踪的信息：

```
> logman start mysession -p "{1cedea2a-a420-5660-1ff0-f718b8ea5138}"
  -o log2.etl -ets
> dotnet run
> logman stop mysession -ets
> tracerpt log2.etl -o log2.xml -of XML
```

服务调用的事件信息如下：

```
<Event xmlns="http://schemas.microsoft.com/win/2004/08/events/event">
  <System>
    <Provider Name="Wrox-SampleEventSource2"
```

```
    Guid="{1cedea2a-a420-5660-1ff0-f718b8ea5138}" />
  <EventID>7</EventID>
  <Version>0</Version>
  <Level>4</Level>
  <Task>0</Task>
  <Opcode>0</Opcode>
  <Keywords>0xF00000000000</Keywords>
  <TimeCreated SystemTime="2017-12-23T13:32:59.015066500+00:59" />
  <Correlation ActivityID="{00000000-0000-0000-0000-000000000000}" />
  <Execution ProcessID="6196" ThreadID="36392" ProcessorID="0"
    KernelTime="30" UserTime="45" />
  <Channel />
  <Computer />
</System>
<EventData>
  <Data Name="url">http://www.cninnovation.com</Data>
</EventData>
<RenderingInfo Culture="en-US">
  <Task>CallService</Task>
</RenderingInfo>
</Event>
```

## 29.2.4 使用注释和 EventSource

创建一个派生于 EventSource 的事件源类，对跟踪信息的定义就有更多的控制。使用特性可以给方法添加注释。

默认情况下，事件源的名字与类名相同，但应用 EventSource 特性，可以改变名字和唯一标识符。每个事件跟踪方法都可以附带 Event 特性。在这里可以定义事件的 ID、操作码、跟踪级别、自定义关键字以及任务。这些信息用来为 Windows 创建清单信息，以定义要记录的信息。方法内使用 EventSource 调用的基本方法 WriteEvent 需要匹配 Event 特性定义的事件 ID，传递给 WriteEvent 方法的变量名需要匹配所声明方法的参数名称。

在示例类 SampleEventSource 中，自定义关键字由内部类 Keywords 定义。这个类的成员强制转换为 EventKeywords 枚举类型。EventKeywords 是基于标识的 long 类型枚举，仅定义高位从 42 开始的值。可以使用所有的低位来定义自定义关键字。Keywords 类为设置为 Network、Database、Diagnostics 和 Performance 的最低四位定义了值。枚举 EventTask 是一个类似的、基于标识的枚举。与 EventKeywords 相反，int 足以用作后备存储，EventTask 没有预定义的值(只有枚举值 None = 0 是预定义的)。类似于 Keywords 类，Task 类为 EventTask 枚举定义了自定义任务 (代码文件 EventSourceSampleAnnotations /SampleEventSource.cs)：

```
class SampleEventSource : EventSource
{
  public class Keywords
  {
    public const EventKeywords Network = (EventKeywords)1;
    public const EventKeywords Database = (EventKeywords)2;
    public const EventKeywords Diagnostics = (EventKeywords)4;
    public const EventKeywords Performance = (EventKeywords)8;
  }

  public class Tasks
  {
    public const EventTask CreateMenus = (EventTask)1;
    public const EventTask QueryMenus = (EventTask)2;
  }

  private SampleEventSource()
  {
  }

  public static SampleEventSource Log = new SampleEventSource ();

  [Event(1, Opcode=EventOpcode.Start, Level=EventLevel.Verbose)]
  public void Startup() => WriteEvent(1);

  [Event(2, Opcode=EventOpcode.Info, Keywords=Keywords.Network,
    Level=EventLevel.Verbose, Message="{0}")]
  public void CallService(string url) => WriteEvent(2, url);
```

```
    [Event(3, Opcode=EventOpcode.Info, Keywords=Keywords.Network,
      Level=EventLevel.Verbose, Message="{0}, length: {1}")]
    public void CalledService(string url, int length) =>
      WriteEvent(3, url, length);

    [Event(4, Opcode=EventOpcode.Info, Keywords=Keywords.Network,
      Level=EventLevel.Error, Message="{0} error: {1}")]
    public void ServiceError(string message, int error) =>
      WriteEvent(4, message, error);

    [Event(5, Opcode=EventOpcode.Info, Task=Tasks.CreateMenus,
      Level=EventLevel.Verbose, Keywords=Keywords.Network)]
    public void SomeTask() => WriteEvent(5);
  }
```

编写这些事件的 Program 类是不变的。这些事件的信息现在可以用于使用侦听器，为特定的关键字、特定的日志级别或特定的任务过滤事件。

## 29.2.5 创建事件清单模式

创建自定义事件源类的优点是，可以创建一个清单，描述所有的跟踪信息。使用没有继承的 EventSource 类，将 Settings 属性设置为枚举 EventSourceSettings 的 EtwSelfDescribingEventFormat 值。事件由所调用的方法直接描述。当使用一个继承自 EventSource 的类时，Settings 属性的值是 EtwManifestEventFormat。事件信息由一个清单描述。

使用 EventSource 类的静态方法 GenerateManifest 可以创建清单文件。第一个参数定义了事件源的类；第二个参数描述了包含事件源类型的程序集的路径 (代码文件 EventSourceSampleAnnotations/Program.cs):

```
public static void GenerateManifest()
{
  string schema = SampleEventSource.GenerateManifest(
    typeof(SampleEventSource), ".");
  File.WriteAllText("sampleeventsource.xml", schema);
}
```

这是包含任务、关键字、事件和事件消息模板的清单信息(代码文件 EventSourceSampleAnnotations/sampleeventsource.xml)：

```
<instrumentationManifest
  xmlns="http://schemas.microsoft.com/win/2004/08/events">
  <instrumentation xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:win="http://manifests.microsoft.com/win/2004/08/windows/events">
    <events xmlns="http://schemas.microsoft.com/win/2004/08/events">
      <provider name="EventSourceSample"
        guid="{45fff0e2-7198-4e4f-9fc3-df6934680096}" resourceFileName="."
        messageFileName="." symbol="EventSourceSample">
        <tasks>
          <task name="CreateMenus" message="$(string.task_CreateMenus)"
            value="1"/>
          <task name="QueryMenus" message="$(string.task_QueryMenus)"
            value="2"/>
          <task name="ServiceError" message="$(string.task_ServiceError)"
            value="65530"/>
          <task name="CalledService" message="$(string.task_CalledService)"
            value="65531"/>
          <task name="CallService" message="$(string.task_CallService)"
            value="65532"/>
          <task name="EventSourceMessage"
            message="$(string.task_EventSourceMessage)" value="65534"/>
        </tasks>
        <opcodes>
        </opcodes>
        <keywords>
          <keyword name="Network" message="$(string.keyword_Network)"
            mask="0x1"/>
          <keyword name="Database" message="$(string.keyword_Database)"
            mask="0x2"/>
          <keyword name="Diagnostics" message="$(string.keyword_Diagnostics)"
            mask="0x4"/>
          <keyword name="Performance" message="$(string.keyword_Performance)"
            mask="0x8"/>
```

```
        <keyword name="Session3" message="$(string.keyword_Session3)"
          mask="0x100000000000"/>
        <keyword name="Session2" message="$(string.keyword_Session2)"
          mask="0x200000000000"/>
        <keyword name="Session1" message="$(string.keyword_Session1)"
          mask="0x400000000000"/>
        <keyword name="Session0" message="$(string.keyword_Session0)"
          mask="0x800000000000"/>
      </keywords>
      <events>
        <event value="0" version="0" level="win:LogAlways"
          symbol="EventSourceMessage" task="EventSourceMessage"
          template="EventSourceMessageArgs"/>
        <event value="1" version="0" level="win:Verbose" symbol="Startup"
          opcode="win:Start"/>
        <event value="2" version="0" level="win:Verbose" symbol="CallService"
          message="$(string.event_CallService)" keywords="Network"
          task="CallService" template="CallServiceArgs"/>
        <event value="3" version="0" level="win:Verbose"
          symbol="CalledService" message="$(string.event_CalledService)"
          keywords="Network" task="CalledService"
          template="CalledServiceArgs"/>
        <event value="4" version="0" level="win:Error" symbol="ServiceError"
          message="$(string.event_ServiceError)" keywords="Network"
          task="ServiceError" template="ServiceErrorArgs"/>
        <event value="5" version="0" level="win:Verbose" symbol="SomeTask"
          keywords="Network" task="CreateMenus"/>
      </events>
      <templates>
        <template tid="EventSourceMessageArgs">
          <data name="message" inType="win:UnicodeString"/>
        </template>
        <template tid="CallServiceArgs">
          <data name="url" inType="win:UnicodeString"/>
        </template>
        <template tid="CalledServiceArgs">
          <data name="url" inType="win:UnicodeString"/>
          <data name="length" inType="win:Int32"/>
        </template>
        <template tid="ServiceErrorArgs">
          <data name="message" inType="win:UnicodeString"/>
          <data name="error" inType="win:Int32"/>
        </template>
      </templates>
    </provider>
  </events>
</instrumentation>
<localization>
  <resources culture="en-US">
    <stringTable>
      <string id="event_CalledService" value="%1 length: %2"/>
      <string id="event_CallService" value="%1"/>
      <string id="event_ServiceError" value="%1 error: %2"/>
      <string id="keyword_Database" value="Database"/>
      <string id="keyword_Diagnostics" value="Diagnostics"/>
      <string id="keyword_Network" value="Network"/>
      <string id="keyword_Performance" value="Performance"/>
      <string id="keyword_Session0" value="Session0"/>
      <string id="keyword_Session1" value="Session1"/>
      <string id="keyword_Session2" value="Session2"/>
      <string id="keyword_Session3" value="Session3"/>
      <string id="task_CalledService" value="CalledService"/>
      <string id="task_CallService" value="CallService"/>
      <string id="task_CreateMenus" value="CreateMenus"/>
      <string id="task_EventSourceMessage" value="EventSourceMessage"/>
      <string id="task_QueryMenus" value="QueryMenus"/>
      <string id="task_ServiceError" value="ServiceError"/>
    </stringTable>
  </resources>
</localization>
</instrumentationManifest>
```

有了这些元数据，通过系统注册它，允许系统管理员过滤特定的事件，在有事发生时得到通知。可以用两种方式处理注册：静态和动态。静态注册需要管理权限，通过 wevtutil.exe 命令行工具注册。该工具传递包含清单的 DLL。EventSource 类也提供了首选的动态注册。这种情况发生在运行期间，不需要管理权限，就可以在

事件流中返回清单，或者回应标准的 ETW 命令。

### 29.2.6 使用活动 ID

TraceSource 新版本的新特性可以轻松地编写活动 ID。一旦运行多个任务，它就有助于了解哪些跟踪消息属于彼此，没有仅基于时间的跟踪消息。例如，对 Web 应用程序使用跟踪时，如果知道哪些跟踪消息属于一个请求，就并发处理多个来自客户端的请求。这样的问题不仅会出现在服务器上，只要运行多个任务，或者使用 C# async 和 await 关键字调用异步方法，这个问题也会出现在客户端应用程序上。此时应使用不同的任务。

当创建派生于 TraceSource 的类时，为了创建活动 ID，只需要定义以 Start 和 Stop 作为后缀的方法。

对于显示活动 ID 的示例，创建一个类库(.NET 标准)。这个库可以在.NET Framework 和.NET Core 应用程序中使用。

.NET 的以前版本不支持活动 ID 的 TraceSource 新功能。ProcessingStart 和 RequestStart 方法用于启动活动；ProcessingStop 和 RequestStop 停止活动(代码文件 MyApplicationEvents/SampleEventSource)：

```
public class SampleEventSource : EventSource
{
  private SampleEventSource()
    : base("Wrox-SampleEventSource") { }

  public static SampleEventSource Log = new SampleEventSource();

  public void ProcessingStart(int x) => WriteEvent(1, x);

  public void Processing(int x) => WriteEvent(2, x);

  public void ProcessingStop(int x) => WriteEvent(3, x);

  public void RequestStart() => WriteEvent(4);

  public void RequestStop() => WriteEvent(5);
}
```

编写事件的客户端应用程序利用如下依赖项和名称空间：

**依赖项**

MyApplicatonEvents

**名称空间**

System

System.Collections.Generic

System.Diagnostics.Tracing

System.Net.Http

System.Threading.Tasks

ParallelRequestSample 方法调用 RequestStart 和 RequestStop 方法来开始和停止活动。在这些调用之间，使用 Parallel.For 创建一个并行循环。Parallel 类通过调用第三个参数的委托，使用多个任务并发运行。这个参数实现为一个 lambda 表达式，来调用 ProcessTaskAsync 方法(代码文件 ClientApp/Program.cs)：

```
private static void ParallelRequestSample()
{
  SampleEventSource.Log.RequestStart();
  Parallel.For(0, 20, async x =>
  {
    await ProcessTaskAsync(x);
  });

  SampleEventSource.Log.RequestStop();
  Console.WriteLine("Activity complete");
}
```

**注意：**

Parallel 类详见第 21 章。

方法 ProcessTaskAsync 使用 ProcessingStart 和 ProcessingStop 写入跟踪信息。在这里，一个活动在另一个活动内部启动。在分析日志的输出中，活动可以带有层次结构(代码文件 ClientApp/Program.cs)：

```
private static async Task ProcessTaskAsync(int x)
{
  SampleEventSource.Log.ProcessingStart(x);
  var r = new Random();
  await Task.Delay(r.Next(500));
  using (var client = new HttpClient())
  {
    var response = await client.GetAsync("http://www.bing.com");
  }
  SampleEventSource.Log.ProcessingStop(x);
}
```

以前，使用 PerfView 工具打开 ETL 日志文件。PerfView 还可以分析运行着的应用程序。可以用以下选项运行 PerfView：

```
> PerfView /onlyproviders=*Wrox-SampleEventSource collect
```

选项 collect 启动数据收集。使用限定符/onlyproviders 关闭内核和 CLR 提供程序，仅记录提供程序列出的日志消息。使用限定符-h 显示可能的选项和 PerfView 的限定符。以这种方式启动 PerfView，会立即开始数据收集，直到单击 Stop Collection 按钮才停止(见图 29-2)。

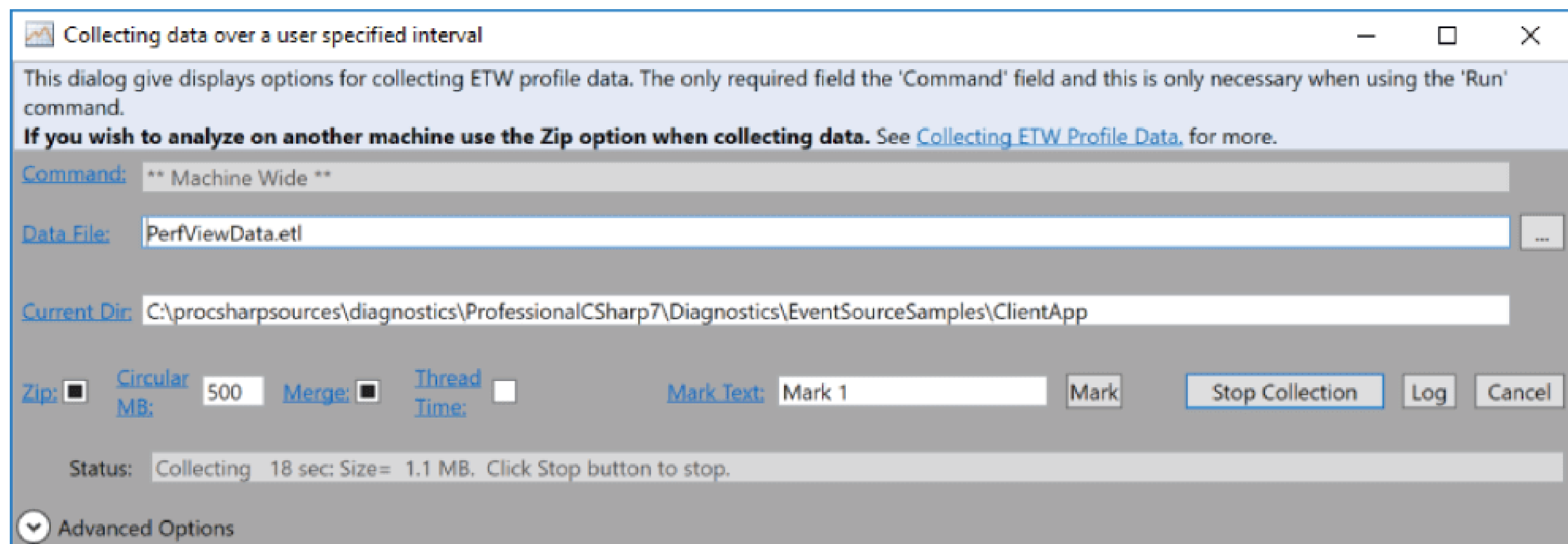



图 29-2

在启动跟踪收集之后运行应用程序，然后停止收集，就可以看到生成的活动 ID 和事件类型 Wrox-SampleEventSource/ProcessingStart/Start。ID 允许有层次结构，例如/ / 1/2 带有一个父活动和一个子活动。每次循环迭代，都会看到一个不同的活动 ID(见图 29-3)。对于事件类型 Wrox-SampleEventSource/ProcessingStop/Stop，可以看到相同的活动 ID，因为它们关联到同样的活动上。

使用 PerfView，可以在左边选择多个事件类型，并添加一个过滤器，例如/ /11/4 ，这样就会看到属于这个活动的所有事件(见图 29-4)。这里可以看到一个活动 ID 可以跨多个线程。相同活动的开始和停止事件使用不同的线程。

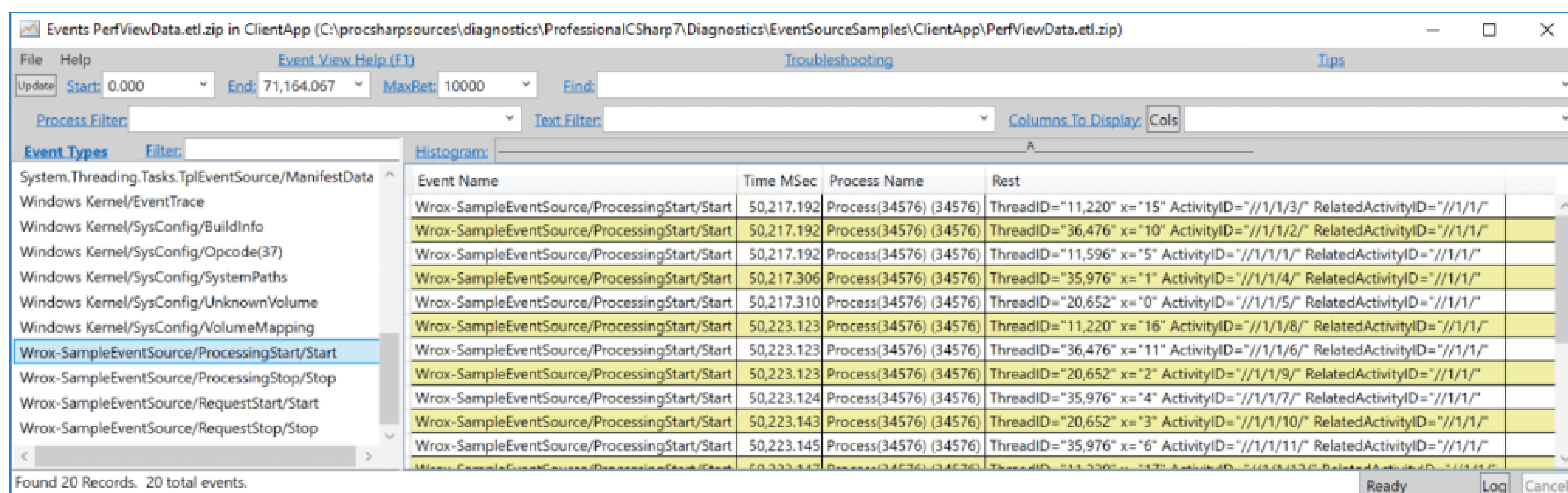



图 29-3

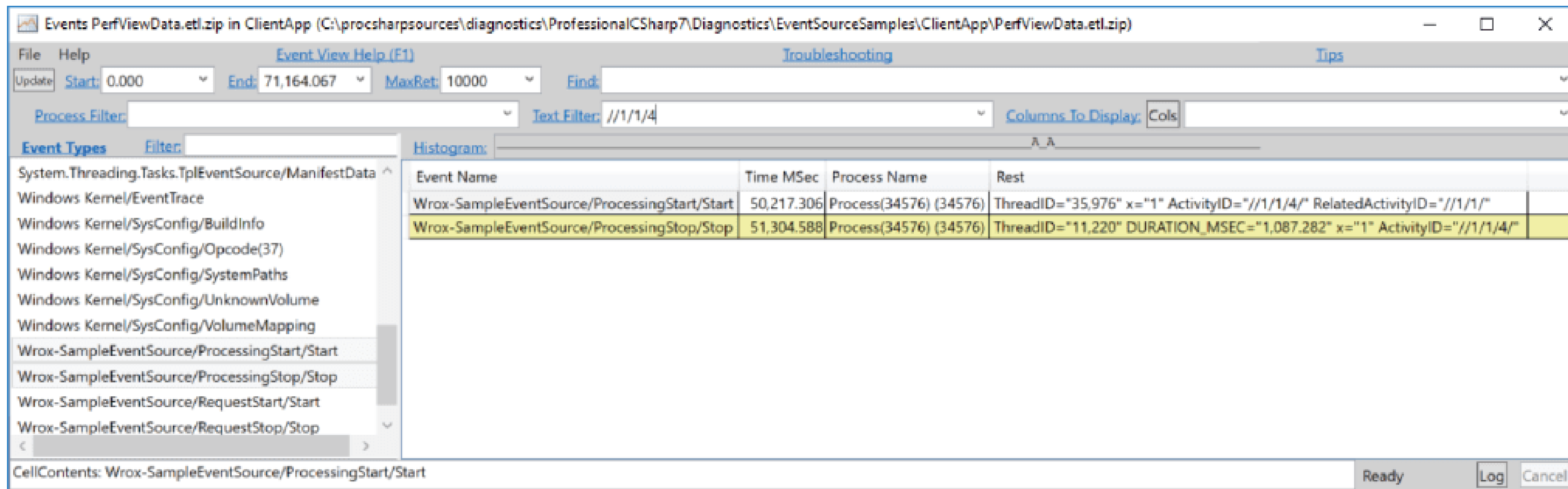


图 29-4

## 29.3 创建自定义侦听器

写入跟踪消息时，我们了解了如何使用工具，如 logman、tracerpt 和 PerfView，读取它们。还可以创建一个自定义的进程内事件侦听器，把事件写入需要的位置。

创建自定义事件侦听器时，需要创建一个派生自基类 EventListener 的类。为此，只需要重写 OnEventWritten 方法。在这个方法中，把跟踪消息传递给类型 EventWrittenEventArgs 的参数。这个样例的实现代码发送事件的信息，包括有效载荷，这是传递给 EventSource 的 WriteEvent 方法的额外数据(代码文件 ClientApp/MyEventListener.cs)：

```
public class MyEventListener : EventListener
{
  protected override void OnEventSourceCreated(EventSource eventSource)
  {
    Console.WriteLine($"created {eventSource.Name} {eventSource.Guid}");
  }

  protected override void OnEventWritten(EventWrittenEventArgs eventData)
  {
    Console.WriteLine($"event id: {eventData.EventId} source: " +
      $" {eventData.EventSource.Name}");
    foreach (var payload in eventData.Payload)
    {
      Console.WriteLine($"\t{payload}");
    }
  }
}
```

侦听器在 Program 类的 Main()方法中激活。通过调用 EventSource 类的静态方法 GetSources，可以访问事件源。

InitListener()方法调用自定义侦听器的 EnableEvents()方法，并传递每个事件源。示例代码注册 EventLevel.LogAlways 设置，来侦听写入的每个日志消息。还可以指定只写入信息性消息，其中还包括错误，或只写入错误。

```
private static void InitListener(IEnumerable<EventSource> sources)
{
  listener = new MyEventListener();
  foreach (var source in sources)
  {
    listener.EnableEvents(source, EventLevel.LogAlways);
  }
}
```

运行应用程序时，会看到 FrameworkEventSource 和 WroxSampleEventSource 的事件写入控制台。使用像这样的自定义事件侦听器，可以轻松地将事件写入 Application Insights，这是一个基于云的遥测服务，参见下一节。

## 29.4　使用 ILogger 接口编写日志

多年来，.NET 中有几种不同的日志记录和跟踪工具，还有许多不同的第三方日志记录程序。尝试将一个应用程序从一种日志记录技术更改为另一种日志记录技术不是一件容易的事情，因为日志记录 API 的使用分布在整个源代码中。要使日志记录独立于任何日志记录技术，可以使用接口。

.NET Core 在 NuGet 包 Microsoft.Extensions.Logging 中嵌入了泛型 ILogger 接口。这个接口定义了 Log 方法。Log 方法定义了参数，来指定 LogLevel(枚举值)、事件 ID(使用结构 EventId)、泛型状态信息、记录异常信息的 Exception 类型，以及用字符串确定输出格式的格式化程序：

```
void Log<TState>(LogLevel logLevel, EventId eventide, TState state,
  Exception exception, Func<TState, Exception, string> formatter);
```

除了 Log 方法之外，ILogger 接口还定义了 IsEnabled 方法，以基于 LogLevel 检查日志记录是否启用，该接口也定义了方法 BeginScope，为日志记录返回可释放的作用域。ILogger 接口中的成员实际上是日志记录所需的全部。Log 方法有许多需要填充的参数。为了简化日志记录，在 LoggerExtensions 类中定义了 ILogger 接口的扩展方法。扩展方法，例如 LogDebug、LogTrace、LogInformation、LogWarning、LogError、LogCritical 和 BeginScope 都有几个重载版本和易于使用的参数。

下面利用依赖注入，并使用包含的类 SampleController 作为一个泛型参数，注入 ILogger 接口。泛型参数定义了日志记录器的类别。在泛型参数中，类别是由类名组成的，包括名称空间(代码文件 LoggingSample/SampleController.cs)：

```
class SampleController
{
  private readonly ILogger<SampleController> _logger;
  public SampleController(ILogger<SampleController> logger)
  {
    _logger = logger;
  }
  //...
}
```

在 29.4.3 节“过滤”中，说明了如何使用类别名来过滤日志。

> **注意：**
> 依赖注入详见第 20 章。

日志示例使用了以下依赖项和名称空间：

**依赖项**

Microsoft.Extensions.DependencyInjection
Microsoft.Extensions.Logging
Microsoft.Extensions.Logging.Configuration
Microsoft.Extensions.Logging.Console
Microsoft.Extensions.Logging.Debug
Microsoft.Extensions.Logging.EventSource
Microsoft.Extensions.Logging.Filter

**名称空间**

Microsoft.Extensions.DependencyInjection
Microsoft.Extensions.Logging
Microsoft.Extensions.Logging.Console
System
System.Net.Http
System.Threading.Tasks

ILogger 接口可以简单地用于调用扩展方法，如 LogInformation：

```
_logger.LogInformation("NetworkRequestSample started");
```

扩展方法提供重载版本，来传递额外的参数、异常信息和事件 ID。为了使用事件 ID，应用程序定义了一个常量值列表(代码文件 LoggingSample/LoggingEvents.cs)：

```
class LoggingEvents
{
  public const int Injection = 2000;
  public const int Networking = 2002;
}
```

接下来，使用 LogInformation 和 LogError 扩展方法显示 NetworkRequestSampleAsync 方法的开头、结束时间以及抛出异常时的错误信息(代码文件 LoggingSample/SampleController.cs)：

```
public async Task NetworkRequestSampleAsync(string url)
{
  try
  {
    _logger.LogInformation(LoggingEvents.Networking,
      "NetworkRequestSampleAsync started with url {0}", url);
    var client = new HttpClient();

    string result = await client.GetStringAsync(url);
    _logger.LogInformation(LoggingEvents.Networking,
      "NetworkRequestSampleAsync completed, received {0} characters",
      result.Length);
  }
  catch (Exception ex)
  {
    _logger.LogError(LoggingEvents.Networking, ex,
      "Error in NetworkRequestSampleAsync, error message: {0}, HResult: {1}",
      ex.Message, ex.HResult);
  }
}
```

**注意：**

ILogger 扩展方法的一个重载版本需要给第一个参数使用 EventId。在示例代码中，传递一个 int。这是可能的，因为 EventId 结构实现了一个隐式运算符，来将 int 转换为 EventId。操作符重载在第 6 章中讨论。

将消息传递给 LogXX 方法时，可以提供任何数量的对象，并将其放入格式消息字符串中。此格式字符串使用位置参数传入以下对象。不能使用可格式化的字符串，因为格式字符串通常来自允许这些消息本地化的资源。本地化在第 27 章中讨论。

接下来，需要配置日志提供程序，以使日志信息可用。

## 29.4.1 配置提供程序

需要使用 ILoggingBuilder 定义配置日志的位置。为 IServiceCollection 调用 AddLogging 扩展方法(这个方法的一个重载版本接受 Action<ILoggingBuilder>参数)时，可以配置 ILoggingBuilder。使用 ILoggingBuilder 时，可以添加提供程序。示例代码为控制台添加提供程序，调试(在 Visual Studio 的 Output 窗口中显示)，并添加事件源代码(代码文件 LoggingSample/Program.cs)：

```
static void RegisterServices()
{
  var services = new ServiceCollection();
  services.AddLogging(builder =>
  {
    builder.AddEventSourceLogger();
    builder.AddConsole();
#if DEBUG
    builder.AddDebug();
#endif
    //...
  });
  services.AddScoped<SampleController>();
  AppServices = services.BuildServiceProvider();
```

```
}

public static IServiceProvider AppServices { get; private set; }
```

示例应用程序的 Main()方法调用 RegisterServices 方法，在依赖注入容器中注册服务，然后调用 RunSampleAsync()方法(代码文件 LoggingSample/Program.cs)：

```
private static string s_url = "https://csharp.christiannagel.com";
static async Task Main(string[] args)
{
  if (args.Length == 1)
  {
    s_url = args[0];
  }
  RegisterServices();
  await RunSampleAsync();
  Console.ReadLine();
}

static async Task RunSampleAsync()
{
  var controller = AppServices.GetService<SampleController>();
  await controller.NetworkRequestSampleAsync(s_url);
}
```

成功运行应用程序时，可以在控制台输出中看到这些信息日志：

```
info: LoggingSample.SampleController[2002]
  NetworkRequestSampleAsync started with url https://csharp.christiannagel.com
info: LoggingSample.SampleController[2002]
  NetworkRequestSampleAsync completed, received 76318 characters
```

传递无效的主机名，会显示如下错误信息：

```
info: LoggingSample.SampleController[2002]
  NetworkRequestSampleAsync started with url https://csharp.christiannagel1.com
fail: LoggingSample.SampleController[2002]
  Error in NetworkRequestSampleAsync, error message: An error occurred
  while sending the request., HResult: -2147012889
```

## 29.4.2　使用作用域

使用作用域，可以将属于输出的日志信息组合在一起。

调用 BeginScope 方法，并将消息传递给作用域，可以定义作用域。该消息显示在输出中，并显示作用域内定义的每个日志消息。BeginScope 返回一个 IDisposable 对象。调用 Dispose 方法(在代码示例中使用 using 语句完成)结束作用域(代码文件 LoggingScopeSample/SampleController.cs)：

```
public async Task NetworkRequestSampleAsync(string url)
{
  using (_logger.BeginScope("NetworkRequestSampleAsync, url: {0}", url))
  {
    try
    {
      _logger.LogInformation(LoggingEvents.Networking, "Started");
      var client = new HttpClient();

      string result = await client.GetStringAsync(url);
      _logger.LogInformation(LoggingEvents.Networking,
        "Completed with characters {0} received", result.Length);
    }
    catch (Exception ex)
    {
      _logger.LogError(LoggingEvents.Networking, ex,
        "Error, error message: {0}, HResult: {1}", ex.Message, ex.HResult);
    }
  }
}
```

对于提供程序，需要启用作用域来显示它。可以更改 AddConsole 方法的配置，来设置 IncludeScopes 属性(代码文件 LoggingScopeSample/Program.cs)：

```
static void RegisterServices()
{
```

```
  var services = new ServiceCollection();
  services.AddLogging(builder =>
  {
    builder.AddEventSourceLogger();
    builder.AddConsole(options => options.IncludeScopes = true);
    builder.AddDebug();
  });
  services.AddScoped<SampleController>();
  AppServices = services.BuildServiceProvider();
}
```

现在运行应用程序时，可以看到在=>之后传递给作用域的信息，如以下代码片段所示：

```
info: LoggingScopeSample.SampleController[2002]
  => NetworkRequestSampleAsync, url: https://csharp.christiannagel.com
  Started
info: LoggingScopeSample.SampleController[2002]
  => NetworkRequestSampleAsync, url: https://csharp.christiannagel.com
  Completed with characters 76395 received
```

### 29.4.3 过滤

不需要在任何时候都查看所有日志消息。应用程序在生产环境中运行时，需要注意错误和关键信息。调试应用程序时，可能会改变配置，为特定的跟踪源显示跟踪消息，了解应用程序中的所有事情。可以为日志记录需求定义过滤器。

过滤可以基于日志记录器提供程序和日志类别。

下面的代码片段为 ConsoleLoggerProvider 和类别名 LoggingSample 定义了一个过滤器，以仅过滤日志级别为 Error 和更高级别的错误(代码文件 LoggingSample/Program.cs)：

```
static void RegisterServices()
{
  var services = new ServiceCollection();
  services.AddLogging(builder =>
  {
    builder.AddEventSourceLogger();
    builder.AddConsole();
#if DEBUG
    builder.AddDebug();
#endif
    builder.AddFilter<ConsoleLoggerProvider>("LoggingSample", LogLevel.Error);
    //...
```

除了指定类别的名称和 LogLevel 之外，还可以传递带有类别和 LogLevel 参数的委托。如果类别名称包含 SampleController，并且所接收的 LogLevel 至少是 Information，那么下面的代码片段将返回一个过滤器值 true。对于所有其他类别，如果 LogLevel 的值至少是 Error，则过滤器返回 true：

```
builder.AddFilter<ConsoleLoggerProvider>((category, logLevel) =>
{
  if (category.Contains("SampleController") &&
    logLevel >= LogLevel.Information) return true;
  else if (logLevel >= LogLevel.Error) return true;
  else return false;
});
```

### 29.4.4 配置日志记录

过滤也可以使用配置文件来定义。

在.NET Core 中，可以给配置文件使用提供程序，例如从 JSON 或 XML 文件、环境变量或命令行参数中读取配置。只需要从 NuGet 包 Microsoft.Extensions.Configuration 中创建一个 ConfigurationBuilder，并向此构建器添加提供程序。要添加 JSON 提供程序，需要调用扩展方法 AddJsonFile。构建器的 Build 方法返回一个实现 IConfiguration 的对象。可以使用此接口通过任何已配置的提供程序来访问已配置值。下面的示例代码从配置中检索 Logging 部分，并将其传递给 RegisterServices 方法(代码文件 LoggingConfigurationSample/Program.cs)：

```
var configurationBuilder = new ConfigurationBuilder();
configurationBuilder.AddJsonFile("appsettings.json");
```

```
IConfiguration configuration = configurationBuilder.Build();
RegisterServices(configuration);
```

示例应用程序的配置文件根据提供程序和类别配置不同的配置值。对于 Debug 提供程序，LogLevel 设置为 Information。这样，对于所有类别，Information 及以上级别都记录到 Visual Studio 的 Output 窗口。对于 Console 提供程序，LogLevel 根据类别的不同而不同。在 Console 提供程序的配置之下，所有其他提供程序的默认配置都是基于类别用特定的日志级别定义(配置文件 LoggingConfigurationSample/appsettings.json)：

```
{
  "Logging": {
    "Debug": {
      "LogLevel": "Information"
    },
    "Console": {
      "LogLevel": {
        "LoggingConfigurationSample.SampleController": "Information",
        "Default": "Warning"
      }
    },
    "LogLevel": {
      "Default": "Warning",
      "System": "Information",
      "LoggingConfigurationSample.SampleController": "Warning"
    }
  }
}
```

在日志配置就绪后，现在调用 AddConfiguration 方法，以传递对 IConfiguration 对象的引用。AddConfiguration 方法需要配置文件中 Logging 部分的内容(代码文件 LoggingConfigurationSample/Program.cs)：

```
static void RegisterServices(IConfiguration configuration)
{
  var services = new ServiceCollection();
  services.AddLogging(builder =>
  {
    builder.AddConfiguration(configuration.GetSection("Logging"))
      .AddConsole();
#if DEBUG
    builder.AddDebug();
#endif
  });
  services.AddScoped<SampleController>();
  AppServices = services.BuildServiceProvider();
}
```

不需要更改任何代码，现在可以灵活地定义日志配置。

**注意：**

Microsoft.Extensions.Configuration 的体系结构和使用不同的配置提供程序详见第 30 章。

### 29.4.5　使用没有依赖注入的 ILogger

依赖注入有很大的优势。但是，也可以使用没有依赖注入的日志记录 API。为此，只需要创建一个 LoggerFactory，并使用 CreateLogger 方法创建一个日志记录器。配置提供程序可以添加到 logger 工厂——类似于为 ILoggingBuilder 接口提供扩展方法，也提供了 ILoggerFactory 的扩展方法(代码文件 LoggingWithoutDI/Program.cs)：

```
var loggerFactory = new LoggerFactory();
loggerFactory.AddConsole().AddDebug();
ILogger<Program> logger = loggerFactory.CreateLogger<Program>();
logger.LogInformation("Info Message");
```

## 29.5 使用 Visual Studio App Center 进行分析

Visual Studio App Center (https://appcenter.ms)是微软开发 Windows 和移动应用程序、向 beta 测试人员分发应用程序、测试应用程序、扩展带有推送通知的应用程序以及获得应用程序的用户分析的入口。

可以得到用户关于应用程序问题的报告，例如，可以找出异常，也可以找到用户在应用程序中正在使用的特性。例如，假设给应用程序添加一个新特性，用户会找到激活该特性的按钮吗？

使用 Application Insights，很容易识别用户使用应用程序时遇到的问题。所以，微软很容易集成 Application Insights 和各种各样的应用程序。

**注意：**

这里有一些特性示例，用户很难在微软自己的产品中找到它们。Xbox 是第一个为用户界面提供大磁贴的设备。搜索特性放在磁贴的下面。虽然这个按钮可以直接显示在用户面前，但用户看不到它。微软把搜索功能移动到磁贴内，现在用户可以找到它。

另一个例子是 Windows Phone 上的物理搜索按钮。这个按钮用于应用程序内的搜索。用户抱怨，没有在电子邮件内搜索的选项，因为他们不认为这个物理按钮可以搜索电子邮件。微软改变了功能。现在物理搜索按钮只用于在网上搜索内容，邮件应用程序有自己的搜索按钮。

Windows 8 有一个相似的搜索问题：用户不使用功能区中的搜索功能，在应用程序内搜索。Windows 8.1 改变了指南，使用功能区中的搜索功能，现在应用程序包含自己的搜索框；在 Windows 10 中还有一个自动显示框。看起来有一些共性？

要启用 app 分析，首先需要注册 Visual Studio App Center。不要担心成本过高——崩溃报告和分析是免费的(在本文撰写时)。接下来，需要创建一个应用程序，并从 Web 门户中复制 App Secret。然后可以用 Visual Studio 创建一个新的 Blank App (Universal Windows)。要启用分析，给项目添加 NuGet 包 Microsoft.AppCenter.Analytics。

只需要使用几个 API 调用，就可以发现用户的问题。在 App 类的构造函数中，添加 AppCenter.Start，并添加先前复制的 App Secret。要启用 Analytics，需要将 Analytics 对象的类型作为第二个参数传递给 Start 方法(代码文件 WinAppAnalytics/App.xam.cs)：

```
public App()
{
  this.InitializeComponent();
  this.Suspending += OnSuspending;

  AppCenter.Start("84df09c4-d560-4c46-a44f-a5524c3abb7f", typeof(Analytics));
}
```

**注意：**

请记住在 Visual Studio App Center 的应用程序配置中，把 App Secret 添加到 Application.Start 方法中。

现在运行应用程序，就会看到用户信息，用户启动应用程序的时间、位置以及来自用户的设备。

要从用户获得更多信息，需要创建对 Analytics.TrackEvent 的调用。应用程序中所有可能的事件都定义在类 EventNames 中(代码文件 WinAppAnalytics/EventNames.cs)：

```
public class EventNames
{
  public const string ButtonClicked = nameof(ButtonClicked);
  public const string PageNavigation = nameof(PageNavigation);
  public const string CreateMenu = nameof(CreateMenu);
}
```

示例应用程序包含一些控件，用于启用/禁用分析、输入一些文本并单击按钮(见图 29-5)。激活 MainPage 时，将收集事件。TrackEvent 方法需要事件名的字符串，该字符串取自 EventNames 类。这个事件的名称不是没有把类名作为前缀，因为用 using static 声明来导入该类的成员。TrackEvent 方法的第二个参数是可选的。在这里，可以传递字符串的一个字典来跟踪其他信息。在示例代码中，当导航到页面时，PageNavigation 事件包含

关于导航到的页面类型的信息(代码文件 WinAppAnalytics/MainPage.xaml.cs)：

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
  base.OnNavigatedTo(e);
  Analytics.TrackEvent(PageNavigation,
    new Dictionary<string, string> { ["Page"] = nameof(MainPage) });
}
```

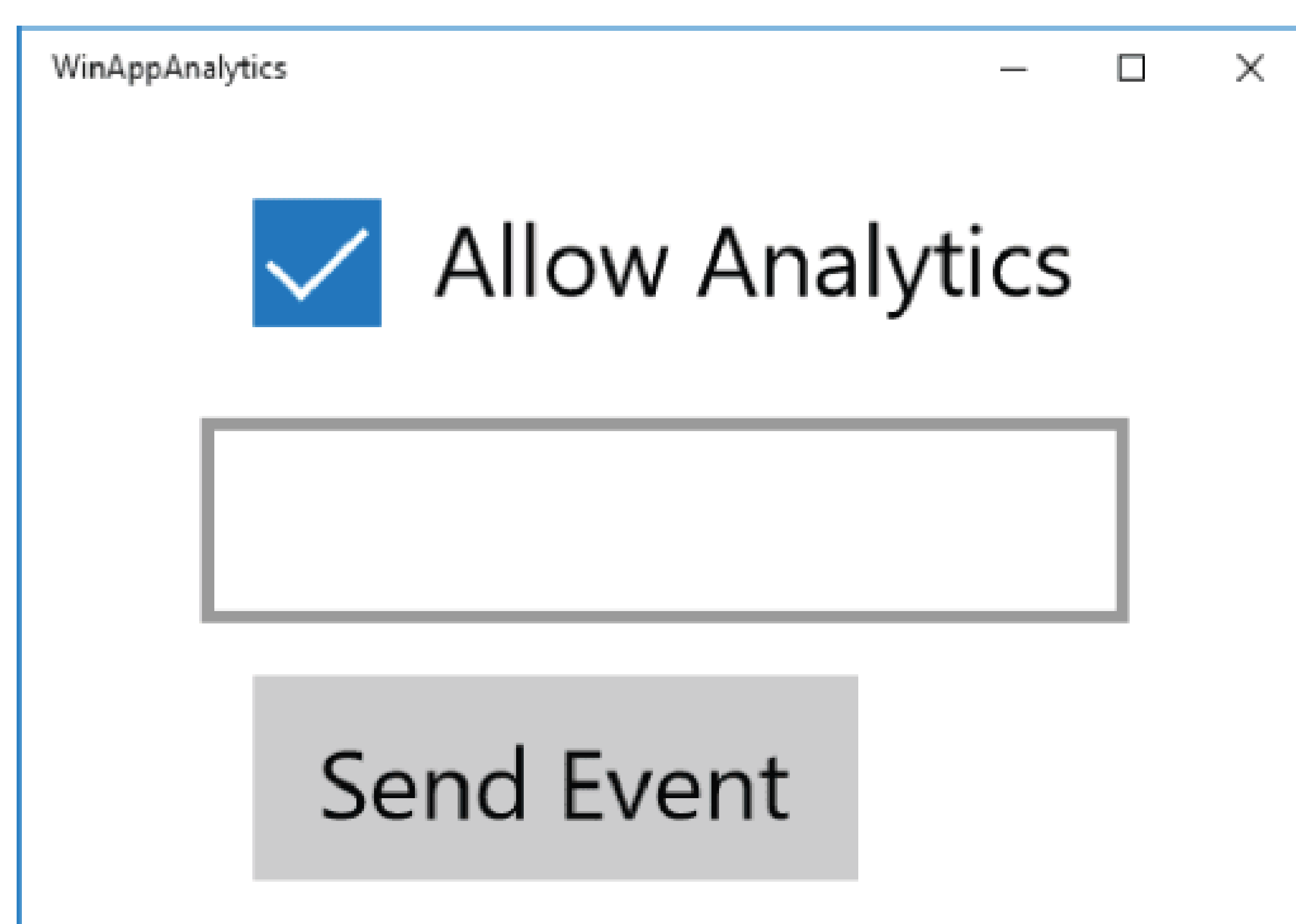



图 29-5

通过单击按钮，TrackEvent 可以跟踪 ButtonClick 事件，用户在 TextBox 控件中输入的信息如下：

```
private void OnButtonClick(object sender, RoutedEventArgs e)
{
  Analytics.TrackEvent(ButtonClicked,
    new Dictionary<string, string> { ["State"] = textState.Text });
}
```

用户在应用程序中漫游时，可能不允许收集信息，因此可以创建一个用户可以用来启用和禁用该功能的设置。如果设置了 Analytics.SetEnabledAsync(false)，那么 Analytics API 将不再报告数据：

```
private async void OnAnalyticsChanged(object sender, RoutedEventArgs e)
{
  if (sender is CheckBox checkbox)
  {
    bool isChecked = checkbox?.IsChecked ?? true;
    await Analytics.SetEnabledAsync(isChecked);
  }
}
```

Visual Studio App Center 在分析方面有一些限制，如下所示：

- 只能有 200 个或更少的事件名称。
- 事件名限制在 256 个字符以内。
- 字典只能包含 5 个或更少的属性。
- 事件属性名称和事件属性值限制在 64 个字符内。

**注意：**
撰写本书时有这些限制。它们可能在未来的版本中改变。

运行应用程序，并监视 Visual Studio App Center 门户时，可以看到发生的事件和受影响的用户数量(参见图 29-6)。单击事件时，可以看到事件计数、每个会话的事件以及传递的字典属性的详细信息。还可以看到实时事件日志流，如图 29-7 所示。

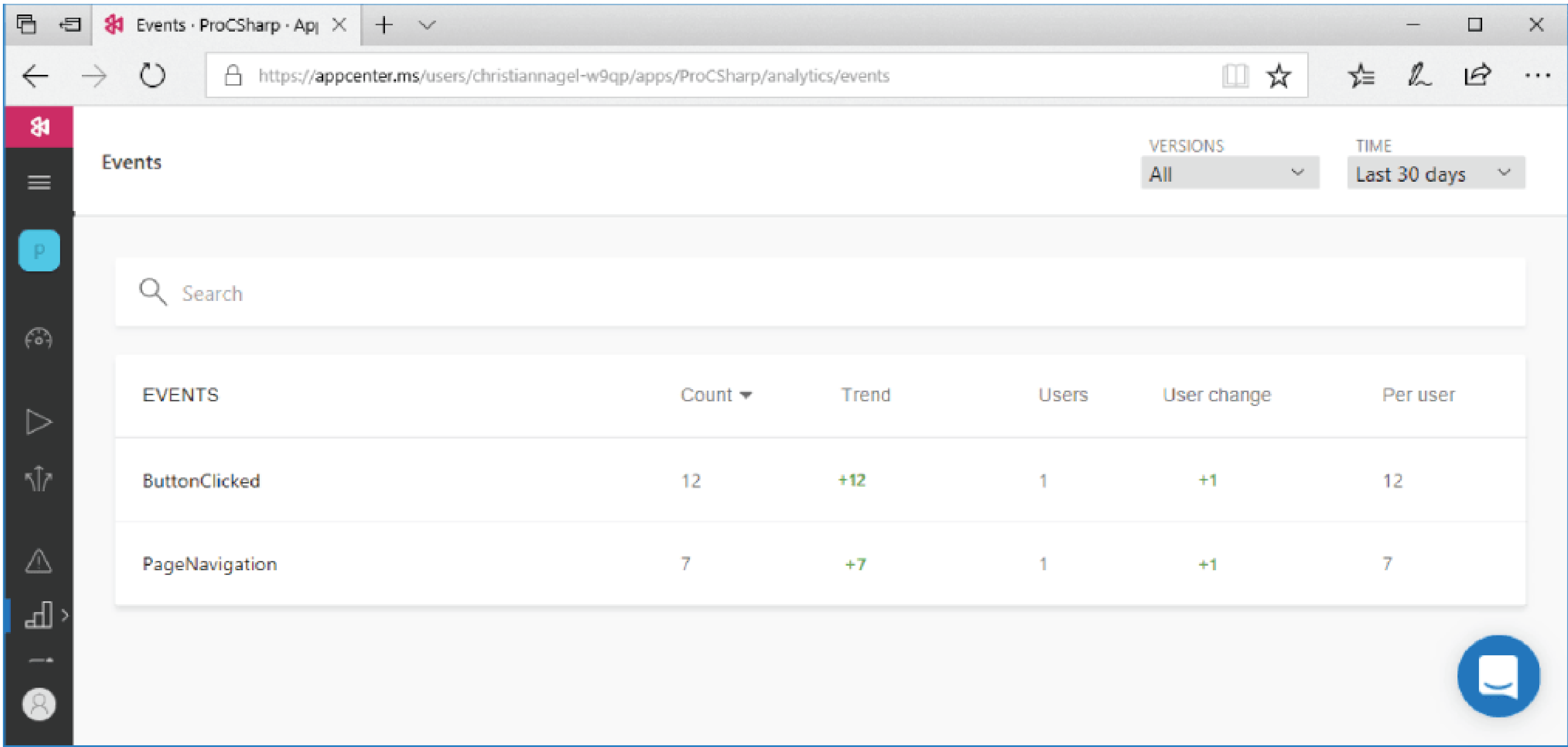



图 29-6

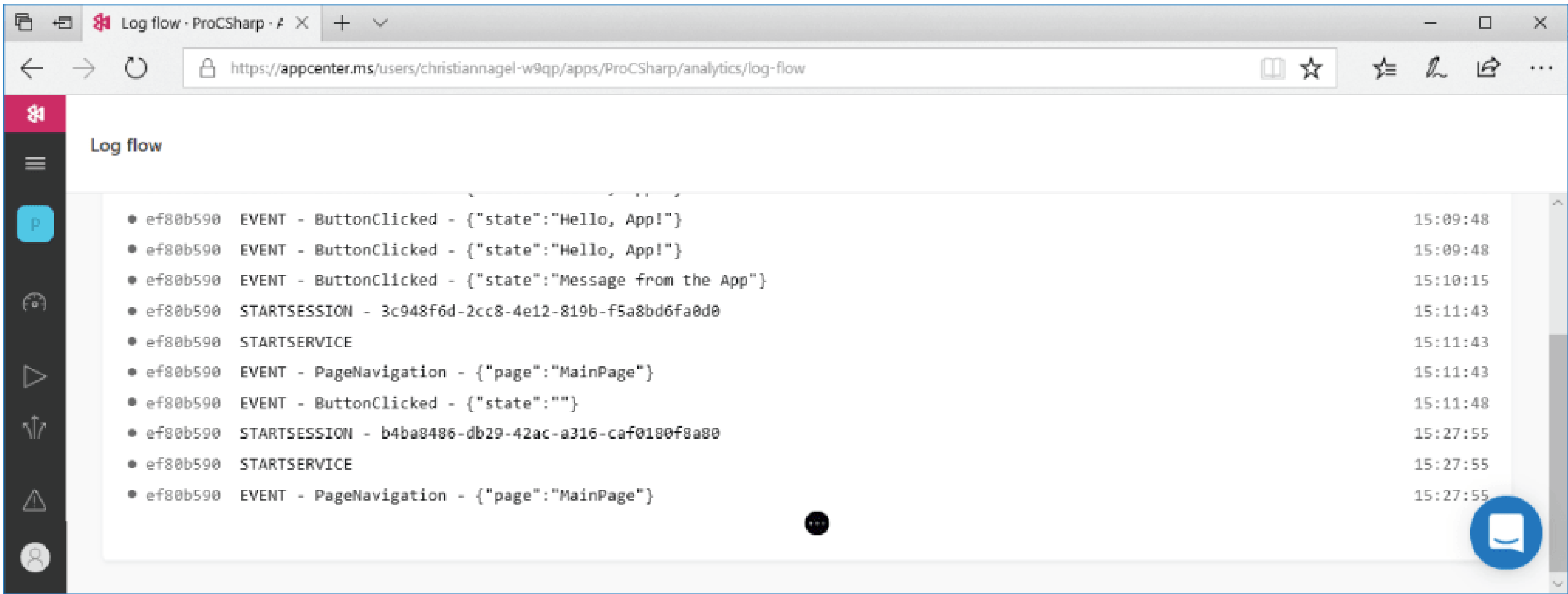



图 29-7

除了这些信息之外，Visual Studio App Center Analytics 还提供了以下信息：

- 活跃用户的数量
- 每个用户每天的会话
- 会话持续时间
- 顶尖设备
- 使用的 OS 版本
- 语言

# 29.6 小结

本章介绍了跟踪和日志功能，它们有助于找出应用程序中的问题。应尽早规划，把这些功能内置于应用程序中。这可以避免以后的许多故障排除问题。

使用跟踪功能，可以把调试消息写入应用程序，也可以用于最终发布的产品。如果出了问题，就可以修改配置值，从而打开跟踪功能，并找出问题。

对于 Visual Studio App Center Analytics，使用这个云服务时，有很多开箱即用的特性可用。只用几行代码，很容易获得用户的信息。如果添加更多代码，可以找到用户是否因为找不到使用应用程序的一些特性而没有使用它们。

这是本书第 II 部分的最后一章。下一部分基于本部分介绍的许多功能。下一章开始探讨 Web 应用程序和服务。

# 第 III 部分

# Web 应用程序和服务

# 第 30 章

# ASP.NET Core

**本章要点**

- 了解 ASP.NET Core 和 Web 技术
- 使用静态内容
- 处理 HTTP 请求和响应
- 使用依赖注入和 ASP.NET
- 定义简单的定制路由
- 创建中间件组件
- 使用会话管理状态
- 读取配置设置

**本章源代码下载地址(wrox.com)：**

打开 www.wrox.com 的 Download Code 选项卡可下载本章源代码。源代码也可以在 aspnetcore 目录的 https://github.com/ProfessionalCSharp/ProfessionalCSharp7 中找到。本章代码包含的示例文件是：

- Simple Host
- WebSampleApp
- CustomConfiguration

## 30.1 概述

在走过 15 年之后，ASP.NET Core 完全重写了 ASP.NET。它的特色在于采用模块化编程，完全开源，是轻量级的，最适合用在云上，可用于非微软平台。

完全重写的 ASP.NET 有很多优势，但这也意味着重写基于老版本 ASP.NET 的现有 Web 应用程序。有必要把现有的 Web 应用程序重写为 ASP.NET Core 版本吗？下面试着回答这个问题。

ASP.NET Web Forms 不再是 ASP.NET Core 的一部分。但是，在 Web 应用程序中包括这项技术并不意味着必须重写它们。仍然可以用完整框架维护用 ASP.NET Web Forms 编写的旧应用程序。在最新版本的.NET Framework 中，ASP.NET Web Forms 甚至有一些增强，如异步的模型绑定。

ASP.NET 的变体 ASP.NET MVC 仍然是 ASP.NET Core 的一部分，但它不同于.NET Framework 的旧框架。对 ASP.NET MVC 和 ASP.NET Core MVC 进行高级比较，会发现这两个技术非常类似。但幕后的所有内容都不同。把 ASP.NET MVC 应用程序转换为 ASP.NET Core MVC，需要对源代码进行一些修改，把它们带到新的应用程序堆栈中。对于一些应用程序，这可能意味着对名称空间、类型和一些方法进行很小的修改。如果应用程序使用了 ASP.NET MVC 中的一些高级功能，就必须进行更多的工作，把应用程序迁移到新技术上。

将 ASP.NET Web Forms 转换为 ASP.NET Core MVC 可能需要做很多工作。ASP.NET Web Forms 从开发人员手中抽象出了 HTML 和 JavaScript。使用 ASP.NET Web Forms，就没有必要了解 HTML 和 JavaScript。只需要使用服务器端控件和 C#代码。服务器端控件返回 HTML 和 JavaScript。此编程模型类似于旧的 Windows Forms 编程模型。使用 ASP.NET MVC，开发人员需要了解 HTML 和 JavaScript。ASP.NET MVC 基于模型-视图-控制器(MVC)模式，便于进行单元测试。因为 ASP.NET Web Forms 和 ASP.NET MVC 基于完全不同的体系结构模式，所以把 ASP.NET Web Forms 应用程序迁移到 ASP.NET MVC 是一个艰巨的任务。承担这个任务之前，应该创建一个清单，列出解决方案仍使用旧技术的优缺点，并与新技术的优缺点进行比较。未来很多年仍可以使用 ASP.NET Web Forms。

**注意：**

网站 http://www.cninnoation.com 最初用 ASP.NET Web Forms 创建。这个用 ASP.NET MVC 早期版本创建的网站被转换到这项新技术堆栈中。因为原来的网站使用了很多独立的组件，抽象出了数据库和服务代码，所以工作量不大，很快就完成了。可以在 ASP.NET MVC 中直接使用数据库和服务。另一方面，如果使用 Web Forms 控件而不是使用自己的控件访问数据库，工作量就很大。目前，这个 Web 应用程序使用 ASP.NET Core 实现。

**注意：**

本书不介绍旧技术 ASP.NET Web Forms，也不讨论 ASP.NET MVC。本书主要论述新技术；因此对于 Web 应用程序，这些内容基于 ASP.NET Core 和 ASP.NET Core MVC。这些技术应该用于新 Web 应用程序。如果需要维护旧应用程序，应该阅读本书的旧版，如《C#高级编程(第 9 版)——C# 5.0 & .NET 4.5.1》，其中介绍了 ASP.NET 4.5、ASP.NET Web Forms 4.5 和 ASP.NET MVC 5。

本章介绍 ASP.NET Core 2.0 的基础知识。第 31 章解释 ASP.NET Core MVC 的用法，这个框架建立在 ASP.NET Core 的基础之上。第 32 章介绍如何用 ASP.NET Core MVC 创建 Web API。

## 30.2 Web 技术

在介绍 ASP.NET Core 的基础知识之前，本节讨论创建 Web 应用程序时必须了解的核心 Web 技术：HTML、CSS、JavaScript 和脚本库。

### 30.2.1 HTML

HTML 是由 Web 浏览器解释的标记语言。它定义的元素显示各种标题、表格、列表和输入元素，如文本框和组合框。

2014 年 10 月以来，HTML5 已经成为 W3C 推荐标准(http://w3.org/TR/html5)，所有主流浏览器都提供了它。在 http://w3.org/TR/html 上有一个工作进度的列表。撰写本书时，HTML5.2 自 2017 年 12 月就有了 W3C 推荐。有了 HTML5 的特性，就不再需要一些浏览器插件(如 Flash 和 Silverlight)了，因为插件可以执行的操作现在都可以直接使用 HTML 和 JavaScript 完成。当然，可能仍然需要 Flash 和 Silverlight，因为不是所有的网站都转而使用新技术，或用户可能仍然使用不支持 HTML5 的旧浏览器版本。

HTML5 添加的新语义元素可以由搜索引擎使用，更好地分析站点。canvas 元素可以动态使用 2D 图形和图像，video 和 audio 元素使 object 元素过时了。由于最近添加的媒体源(http://w3c.github.io/media-source)，自适应流媒体也由 HTML 提供；此前这是 Silverlight 的一个优势。

HTML5 还为拖放操作、存储器、Web 套接字等定义了 JavaScript API。

### 30.2.2 CSS

HTML 定义了 Web 页面的内容，CSS 定义了其外观。例如，在 HTML 的早期，列表项标记<li>定义列表元素在显示时是否应带有圆、圆盘或方框。目前，这些信息已从 HTML 中完全删除，而放在 CSS 中。

在 CSS 样式中，HTML 元素可以使用灵活的选择器来选择，还可以为这些元素定义样式。元素可以通过其 ID 或名称来选择，也可以定义 CSS 类，从 HTML 代码中引用。在 CSS 的新版本中，可以定义相当复杂的规则，来选择特定的 HTML 元素。

自 Visual Studio 2017 起，一些 Web 项目模板使用 Twitter Bootstrap，这是 CSS 和 HTML 约定的集合。这就易于采用不同的外观，下载易用的模板。文档和基本模板可参阅 www.getbootstrap.com。

### 30.2.3 JavaScript 和 TypeScript

并不是所有的平台和浏览器都能使用.NET 代码，但几乎所有的浏览器都能理解 JavaScript。对 JavaScript 的一个常见误解是它与 Java 相关。实际上，它们只是名称相似，因为 Netscape(JavaScript 的发起者)与 Sun(Sun 发明了 Java)达成了协议，允许在名称中使用 Java。如今，这两个公司不再存在。Sun 被 Oracle 收购，现在 Oracle 持有 Java 的商标。

Java 和 JavaScript 有相同的根(C 编程语言)。JavaScript 是一种函数式编程语言，不是面向对象的，但它添加了面向对象功能。

JavaScript 允许从 HTML 页面访问 DOM(Document Object Model，文档对象模型)，因此可以在客户端动态改变元素。

ECMAScript 是一个标准，它定义了 JavaScript 语言的当前和未来功能。因为其他公司在其语言实现中不允许使用 Java 这个词，所以该标准的名称是 ECMAScript。Microsoft 的 JavaScript 实现被命名为 JScript。访问 https://tc39.github.io/ecma262/，可了解 JavaScript 语言的当前状态和未来的变化。

尽管许多浏览器不支持最新的 ECMAScript 版本，但仍然可以编写 ECMAScript 2018 代码。不是编写 JavaScript 代码，而是可以使用 TypeScript。TypeScript 语法基于 ECMAScript，但是它有一些改进，如强类型代码和注解。C#和 TypeScript 有很多相似的地方。因为 TypeScript 编译器编译成 JavaScript，所以 TypeScript 可以用在需要 JavaScript 的所有地方。有关 TypeScript 的更多信息可访问 http://www.typescriptlang.org。

### 30.2.4 脚本库

除了 JavaScript 编程语言之外，还需要脚本库简化编程工作。脚本库可以与 ASP.NET Core 的服务器端功能一起使用：

- jQuery(http://www.jquery.org)是一个库，它抽象出了访问 DOM 元素和响应事件时的浏览器的差异。几年前，这个库应用于几乎每个网站。但目前，有了更多的选项，jQuery 不会应用于所有地方了。
- Angular(https://angular.io)是 Google 中一个基于 MVC 模式的库，用单页面的 Web 应用程序简化了开发和测试(与 ASP.NET MVC 不同，Angular 提供了 MVC 模式与客户端代码)。
- React (https://reactjs.org)是来自 Facebook 的一个库，提供的功能便于在数据改变时在后台更新用户界面。

用于 Visual Studio 的 ASP.NET Core 2.0 项目模板包括 Angular 和 React。Visual Studio 2017 支持智能感知和对 JavaScript 代码的调试。

**注意：**

本书未涉及指定 Web 应用程序的样式和编写 JavaScript 代码。关于 HTML 和样式，可以参阅 John Duckett 编著的《HTML & CSS 设计与构建网站》(John Wiley & Sons, 2011)；进而阅读 Jeremy McPeak 编著的 *Beginning JavaScript，Fifth Edition*(Wrox, 2015)。

## 30.3　ASP.NET Web 项目

首先创建一个空的 ASP.NET Core 2.0 Web 应用程序。第一个应用程序是一个简单的主机，它只响应请求。从一个新的 ASP.NET Core Web 应用程序开始，并选择空模板(参见图 30-1)。但是对于第一个示例，空模板空得还不够。从模板中删除 Startup.cs 文件和 wwwroot 目录。

Main()方法简化为调用 WebHost 类的 Start()方法。此方法具有 RequestDelegate 参数。RequestDelegate 是一个委托，把 HttpContext 接收为参数并返回一个 Task。可以使用 HttpContext 从客户端读取请求并发送返回的内容。使用示例代码，返回包含 HTML 字符串的响应(代码文件 SimpleHost/Program.cs)：

```
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;

namespace SimpleHost
{
  public class Program
  {
    public static void Main()
    {
      WebHost.Start(async context =>
      {
        await context.Response.WriteAsync("<h1>A Simple Host!</h1>");
      }).WaitForShutdown();
    }
  }
}
```

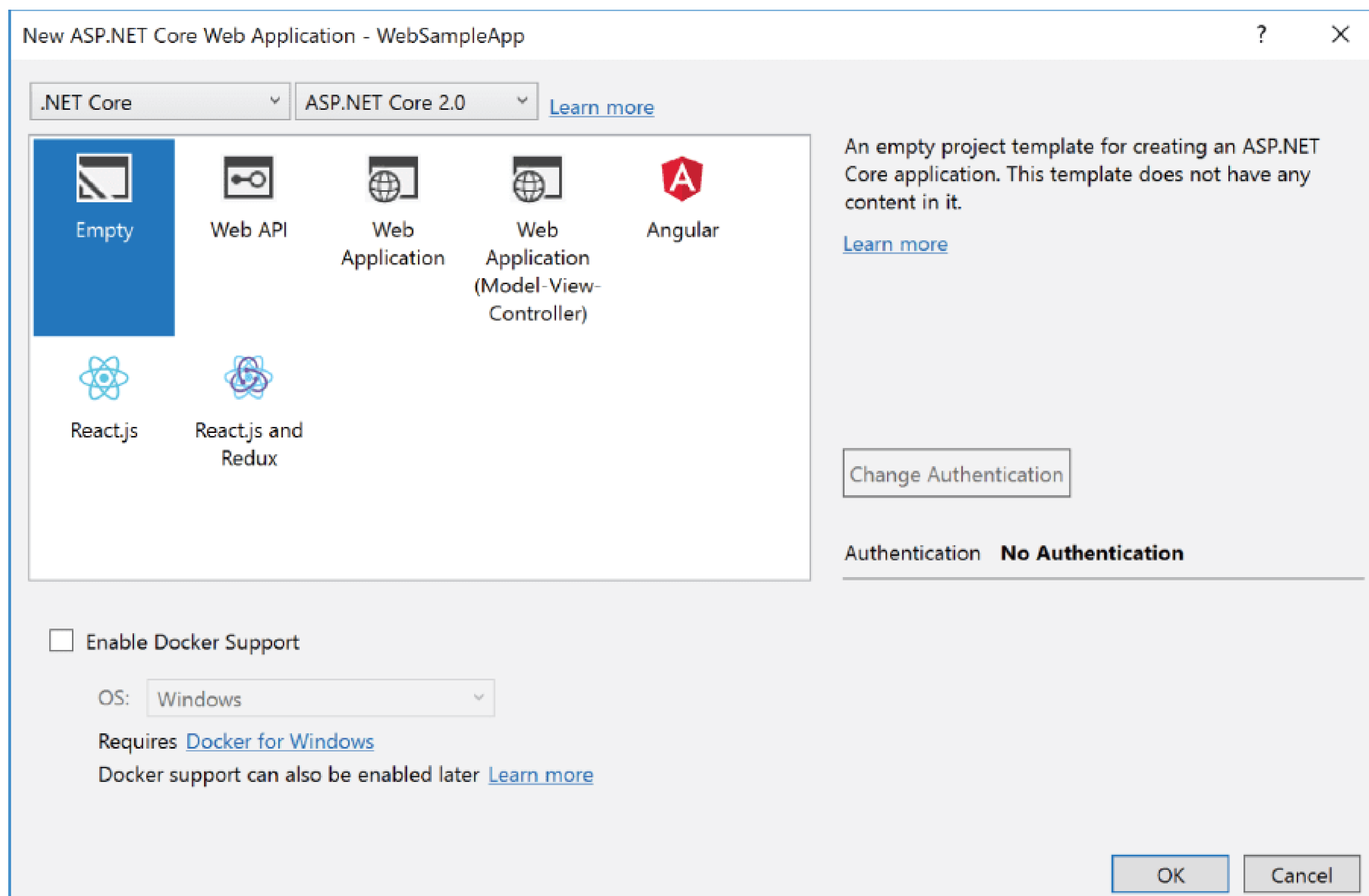



图 30-1

运行应用程序时，可以在浏览器中看到 HTML 内容。

使用 ASP.NET Core 创建 Web 主机非常简单，但是现在进入一个更复杂的场景来看看这些特性。下一个应用程序名为 WebSampleApp，使用相同的 Empty 模板创建。

创建项目之后，会得到一个名为 WebSampleApp 的解决方案和一个项目文件，其中包括一些文件和文件夹(参见图 30-2)。

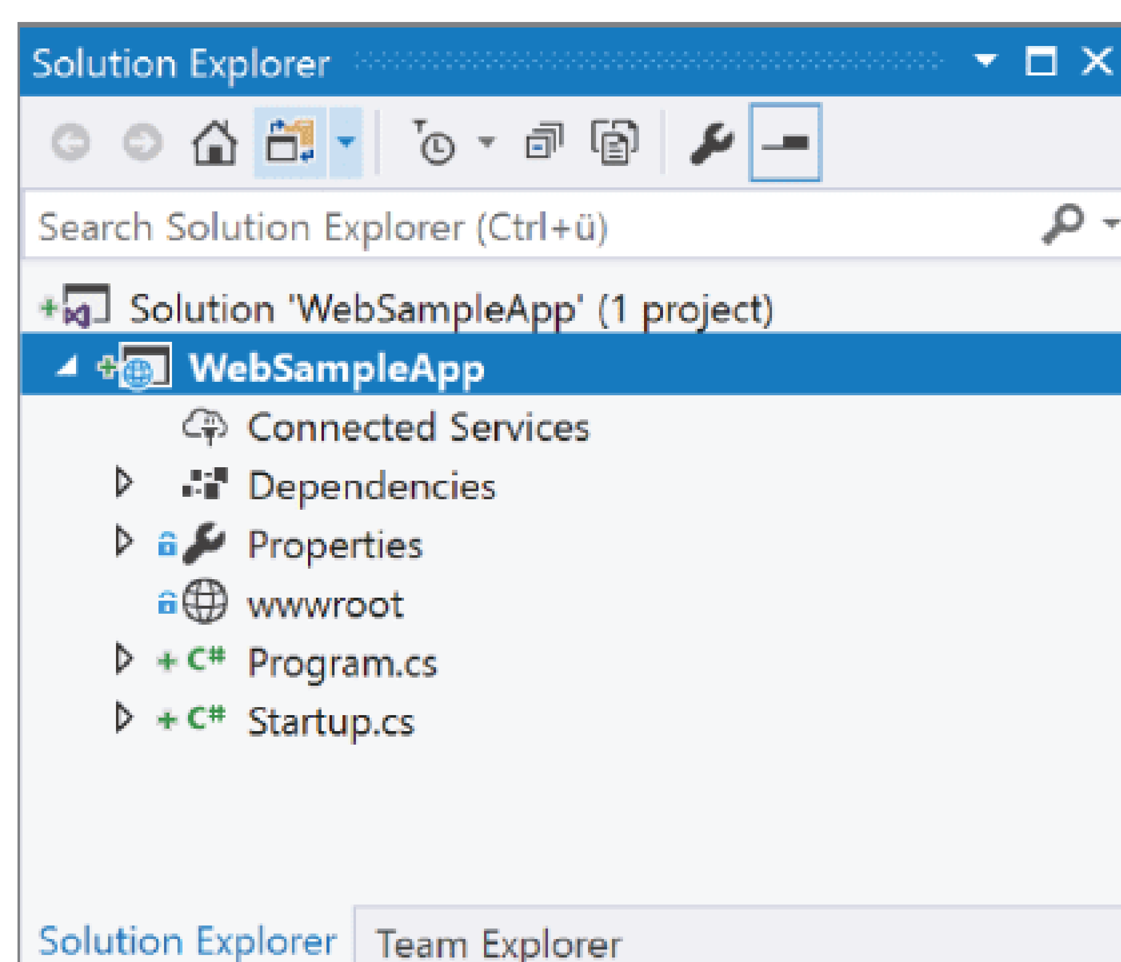



图 30-2

在项目结构中，有一个 Dependencies 文件夹。其中的 NuGet 子文件夹包含 NuGet 包。在 ASP.NET Core 2.0，包列表已经简化，只能看到 Microsoft.AspNetCore.All 引用包。这是一个包含大量 ASP.NET Core 包的引用包。在 Solution Explorer 中打开 Microsoft.AspNetCore.All 时引用的包列表。

在项目文件中，还可以看到对这个包的引用。项目文件列出了项目 SDK(软件开发工具包)和 Microsoft.NET.Sdk.Web。这利用了安装在系统上的 SDK。这个条目不同于控制台应用程序，其中 SDK 是 Microsoft.NET.Sdk。在 Web SDK 中，可以使用其他 Web 开发工具(项目文件 WebSampleApp.cspoj)：

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <Folder Include="wwwroot\" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
  </ItemGroup>

</Project>
```

在 Project 设置中使用 Debug 选项，可以配置提供 Visual Studio 开发时使用的 Web 服务器(参见图 30-3)。在默认情况下，IIS Express 配置为使用 Debug 设置指定的端口号。IIS Express 源自 Internet Information Server (IIS)，提供了 IIS 的所有核心特性。所以非常易于在与稍后托管应用程序的环境(如果 IIS 用于托管)几乎相同的环境中开发 Web 应用程序。

要使用 Kestrel 服务器运行应用程序，可以使用 Debug Project 设置选择项目名称的概要文件。使用 Visual Studio 项目设置更改的设置将影响 launchSettings.json 文件的配置。通过这个文件，可以定义一些附加的配置，比如命令行参数(代码文件 WebSampleApp/Properties/launchsettings.json)：

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:19879/",
```

```
            "sslPort": 0
          }
        },
        "profiles": {
          "IIS Express": {
            "commandName": "IISExpress",
            "launchBrowser": true,
            "environmentVariables": {
              "Hosting:Environment": "Development"
            }
          },
          "web": {
            "commandName": "web",
            "launchBrowser": true,
            "launchUrl": "http://localhost:5000/",
            "commandLineArgs": "Environment=Development",
            "environmentVariables": {
              "Hosting:Environment": "Development"
            }
          }
        }
      }
```

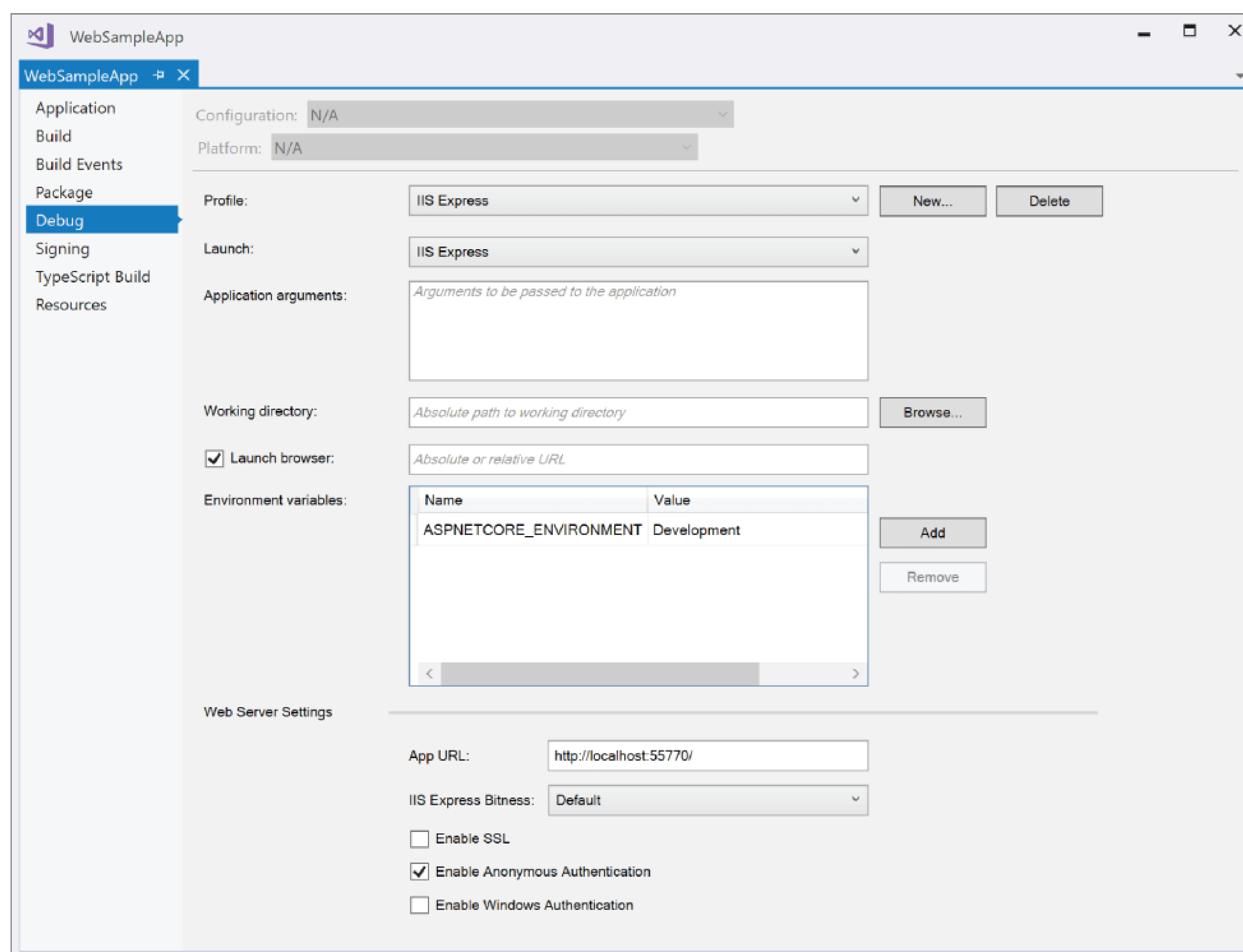



图 30-3

**注意：**

Kestrel 服务器是由 ASP.NET Core 团队开发，通过 ASP.NET Core 提供简单的主机。当使用 IIS 托管 Web 应用程序时，IIS 将请求转发给 Kestrel 服务器。这就像 Web 应用程序在 Linux 上由 Apache 服务器托管一样——将请求转发到 Kestrel 服务器。首先是对于 ASP.NET Corer 2.0，Kestrel 服务器支持面向公众的使用，因此可以直接在 Kestrel 服务器上托管 Web 应用程序，并且可以从端口 80 上访问它。

Solution Explorer 的项目结构中的 Dependencies 文件夹显示了对 JavaScript 库的依赖。创建空项目时，这个文件夹是空的。30.5 节“添加静态内容”会添加依赖项。

wwwroot 文件夹包含了需要发布到服务器的静态文件。目前，这个文件夹是空的，但是将添加 HTML、CSS 文件和 JavaScript 库。

C#源文件 Startup.cs 也包含在一个空项目中。接下来将讨论这个文件。

在项目的创建过程中，需要如下名称空间：

Microsoft.AspNetCore.Builder;
Microsoft.AspNetCore.Hosting;
Microsoft.AspNetCore.Http;
Microsoft.Extensions.Configuration
Microsoft.Extensions.DependencyInjection
Microsoft.Extensions.Logging
Microsoft.Extensions.PlatformAbstractions
Newtonsoft.Json
System
System.Collections.Generic
System.Globalization
System.Linq
System.Text
System.Text.Encodings.Web
System.Threading.Tasks

## 30.3.1　启动

下面开始建立 Web 应用程序的一些功能。为了获得有关客户端的信息并返回一个响应，需要编写对 HttpContext 的响应。

使用空的 ASP.NET Web 应用程序模板在 Program 类中创建一个 Main()方法，其中包含以下代码(代码文件 WebSampleApp/Program.cs)：

```
public class Program
{
  public static void Main(string[] args)
  {
    BuildWebHost(args).Run();
  }

  public static IWebHost BuildWebHost(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
      .UseStartup<Startup>()
      .Build();
}
```

CreateDefaultBuilder 返回一个实现 IWebHostBuilder 的对象，该对象设置了以下功能：

- 配置 Kestrel 作为要使用的 Web 服务器。
- 内容的根路径设置为当前目录。
- 将配置定义为从 appsettings.json 文件中加载的配置。
- 基于环境，用不同的名称添加一个附加的 JSON 配置文件

```
appsettings.{environmentname}.json
```

- 环境名称是 Development 时，将读取来自 user secrets 的配置。
- 将配置设置提供程序配置为从环境变量和命令行中加载设置。
- 将记录器工厂配置为记录到控制台和调试输出窗口上。
- 启用 IIS 集成。

**注意：**
用户机密参见第 24 章。

使用从 CreateDefaultBuilder 返回的 IWebHostBuilder 时，将调用 UseStartup 方法。该方法定义要实例化的 Startup 类和运行 Web 主机时将调用的方法。

UseStartup方法实现为一个流利API，并再次返回IWebHostBuilder。可以在Startup类之前使用IWebHostBuilder来配置需要的其他服务，并定义其他配置提供程序。

Build 方法是设置 Web 主机的链中的最后一个方法。此方法构建主机来运行应用程序，并返回 IWebHost 接口。使用此接口时，可以使用 Services 属性访问依赖注入容器，也可以从 ServerFeatures 属性中访问托管服务器特性。IServerAddressesFeature 是一个可以用来检索主机地址的服务器特性。调用 Start 方法将启动对已配置端口的套接字的监听。

**注意：**
依赖注入和.NET Core 依赖注入容器 Microsoft.Extensions.DependencyInjection 详见第 20 章。

空的 ASP.NET Web 应用程序模板创建一个 Startup 类，它包含以下代码：

```
//...
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace WebSampleApp
{
  public class Startup
  {
    public void ConfigureServices(IServiceCollection services)
    {
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
      if (env.IsDevelopment())
      {
        app.UseDeveloperExceptionPage();
      }

      app.Run(async (context) =>
      {
        await context.Response.WriteAsync("Hello World!");
      });
    }
  }
}
```

由于使用泛型模板参数将 Startup 类传递给 UseStartup 方法，因此将调用 ConfigureServices 和 Configure 方法。

可以使用 ConfigureServices 方法在依赖注入容器中配置服务。此方法具有 IServiceCollection 属性，该属性包含 Main()方法中已注册的所有服务，并允许添加其他服务。IServiceCollection 派生自基接口 IList<t>，使用 ServiceDescriptor 作为泛型参数，因此不仅允许读取服务，还允许添加服务。

Configure()方法通过依赖注入接收参数。模板中定义的参数是 IApplicationBuilder 类型和 IHostingEnvironment 类型。

接口 IHostingEnvironment 允许访问环境的名称(EnvironmentName)、内容的根路径(源代码的目录)和 Web 内容文件的根路径(子目录 wwwroot)。访问这些目录的默认提供程序是 PhysicalFileProvider。对于不同的提供程序，可以从其他数据源(例如数据库)中提供内容。在 Configure 方法的实现中，使用 IHostingEnvironment 通过调用扩展方法 IsDevelopment 来检查当前环境是否是 Development。只有在这个环境中才显示异常。由于安全问题，在生产环境中，用户看不到异常的详细信息。

IApplicationBuilder 接口用于向 HTTP 请求管道添加中间件。调用这个接口的 Use 方法时，可以构建 HTTP 请求管道，来定义响应请求时应该做什么。Use 方法是使用流利 API 实现的，它再次返回 IApplicationBuilder。

这样，可以很容易地将多个中间件对象添加到管道中。有几种扩展方法可以使添加中间件更加容易。在本章后面可以创建自定义中间件并将其添加到管道中。

Run 方法是接口 IApplicationBuilder 的扩展方法，并返回 void。因此，它在请求管道中注册最后一个中间件。Run 方法的参数是 RequestDelegate 类型的委托。该类型接收 HttpContext 作为参数，并返回一个 Task。使用 HttpContext(代码片段中的 context 变量)，可以访问来自浏览器的请求信息(HTTP 标题、cookie 和表单数据)，并可以发送响应。生成的代码给客户端返回一个简单的字符串—Hello World!，如图 30-4 所示。

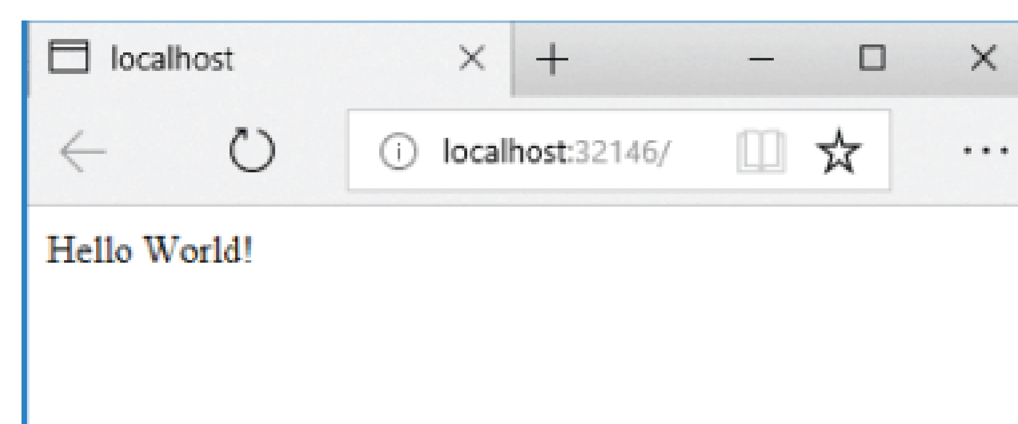



图 30-4

**注意：**

如果使用 Microsoft Edge 测试 Web 应用程序，就需要启用 localhost。在 URL 框中输入 about:flags，启用 Allow localhost loopback 选项(参见图 30-5)。除了使用 Microsoft Edge 内置的用户界面设置此选项之外，还可以使用命令行选项：实用工具 CheckNetIsolation。命令：

```
CheckNetIsolation LoopbackExempt -a -n=Microsoft.MicrosoftEdge _&wekyb3d8bbwe
```

可以启用 localhost，类似于使用 Microsoft Edge 中更友好的用户界面。如果想配置其他 Windows 应用程序以启用 localhost，也可以使用实用程序 CheckNetIsolation。

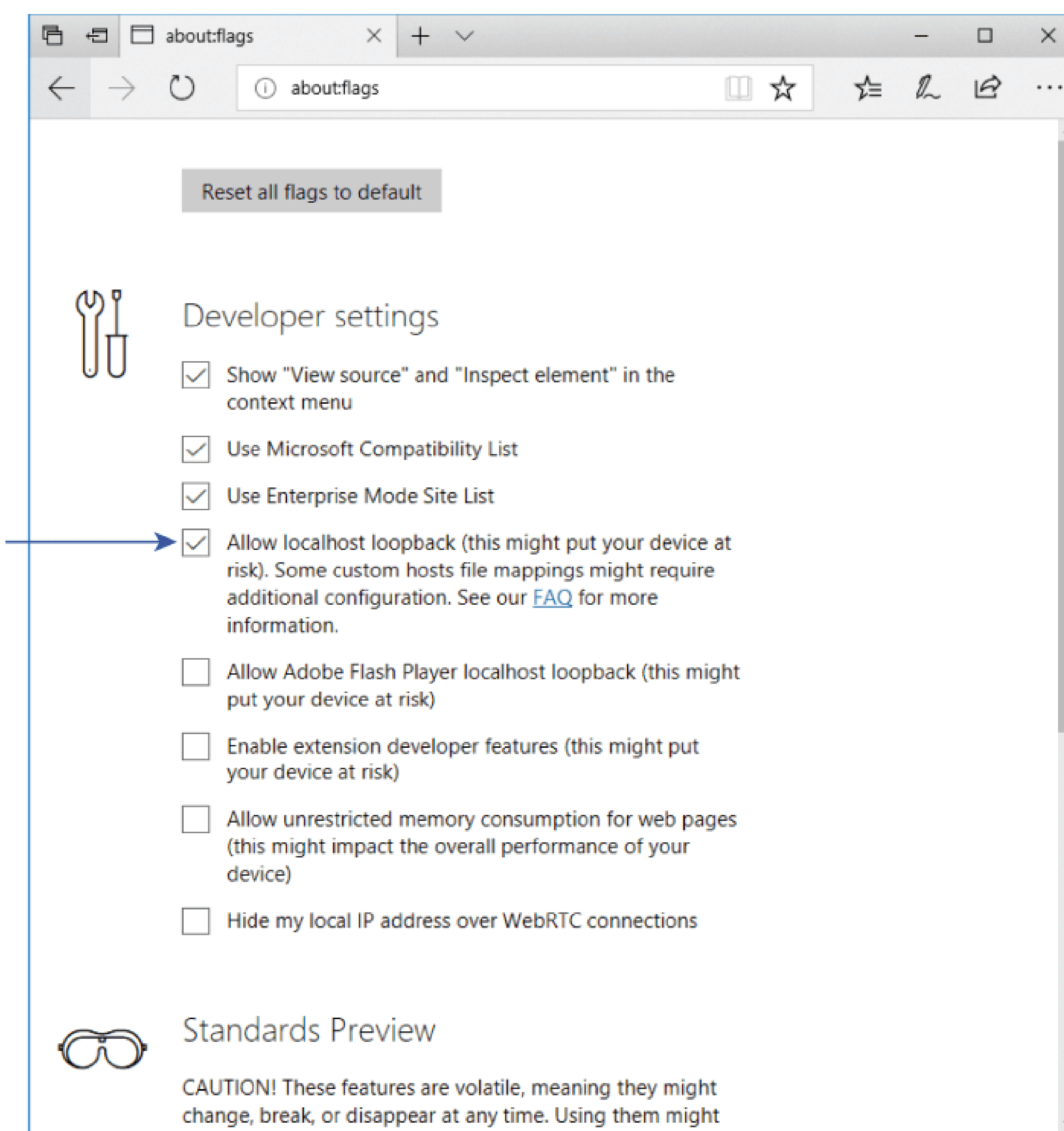



图 30-5

## 30.3.2 示例应用程序

示例应用程序包含一个入口页面，在该页面中，可以使用 HTML 链接轻松访问应用程序显示的所有特性：

```
app.Run(async (context) =>
{
  string[] lines = new[]
  {
    @"<ul>",
      @"<li><a href=""/hello.html"">Static Files</a> - requires " +
        @"UseStaticFiles</li>",
      @"<li>Request and Response",
        @"<ul>",
          @"<li><a href=""/RequestAndResponse"">Request and Response</a></li>",
          @"<li><a href=""/RequestAndResponse/header"">Header</a></li>",
          @"<li><a href=""/RequestAndResponse/add?x=38&y=4"">Add</a></li>",

          //...

        @"</ul>",
      @"</li>",
    @"</ul>"
  };

  var sb = new StringBuilder();
  foreach (var line in lines)
  {
    sb.Append(line);
  }
  string html = sb.ToString().HtmlDocument("Web Sample App");

  await context.Response.WriteAsync(html);
});
```

定义 HTMLExtensions 类是为了创建特定的 HTML 并减少需要编写的 HTML 代码。这个类定义扩展方法来创建 div、span 和 li 元素(代码文件 WebSampleApp/HtmlExtensions.cs)：

```
public static class HtmlExtensions
{
  public static string Div(this string value) =>
    $"<div>{value}</div>";

  public static string Span(this string value) =>
    $"<span>{value}</span>";

  public static string Div(this string key, string value) =>
    $"{key.Span()}: {value.Span()}".Div();

  public static string Li(this string value) =>
    $@"<li>{value}</li>";

  public static string Li(this string value, string url) =>
    $@"<li><a href=""{url}"">{value}</a></li>";

  public static string Ul(this string value) =>
    $"<ul>{value}</ul>";

  public static string HtmlDocument(this string content, string title)
  {
    var sb = new StringBuilder();
    sb.Append("<!DOCTYPE HTML>");
    sb.Append("<head><meta charset=\"utf-8\"><title>{title}</title></head>");
    sb.Append("<body>");
    sb.Append(content);
    sb.Append("</body>");
    return sb.ToString();
  }
}
```

# 30.4 添加客户端内容

通常不希望只把简单的字符串发送给客户端。默认情况下，不能发送简单的 HTML 文件和其他静态内容。ASP.NET Core 会尽可能减少开销。如果没有启用，即使是静态文件也不能从服务器返回。

要在 Web 服务器上处理静态文件，可以添加扩展方法 UseStaticFiles，以添加需要的中间件(代码文件 WebSampleApp/Startup.cs)：

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
  /...
  app.UseStaticFiles();

  app.Run(async (context) =>
  {
    await context.Response.WriteAsync("Hello World!");
  });
}
```

添加静态文件的文件夹是项目内的 wwwroot 文件夹。下面将一个简单的 HTML 文件添加到 wwwroot 文件夹中，以添加静态内容(代码文件 WebSampleApp/wwwroot/Hello.html)，如下所示：

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>ASP.NET Core Sample</title>
  </head>
  <body>
    <h1>Hello, ASP.NET with Static Files</h1>
  </body>
</html>
```

现在，启动服务器后，从浏览器中向 HTML 文件发出请求，例如 http://localhost:5000/Hello.html。根据正在使用的配置，项目的端口号可能会有所不同。如果去掉了扩展方法 UseStaticFiles 的注释符号，HTML 文件就不从请求中返回。

**注意：**

用 ASP.NET Core 创建 Web 应用程序时，还需要了解 HTML、CSS、JavaScript 和一些 JavaScript 库。本书的重点是 C#和.NET Core，所以这些主题的内容非常少。本书仅讨论使用 ASP.NET Core 时需要知道的最重要的任务。

### 30.4.1 为客户端内容使用工具

要为客户端创建内容，还需要一些工具。使用老版本的 ASP.NET，一切都集成在 Visual Studio 中。可以使用 NuGet 包下载并安装 JavaScript 库。但是，由于关注脚本库的社区通常不使用 NuGet 服务器，所以社区也不为 JavaScript 库创建 NuGet 包。关注 JavaScript 库的社区使用具有 NuGet 等功能的服务器，而不是使用 NuGet。

微软和 NuGet 社区为 JavaScript 库构建 NuGet 包，以便在 Visual Studio 中使用。这总是会产生一些延迟，通常使用 Visual Studio 的体验并不是 Web 世界中最好的。

许多开发 Web 应用程序的工具和库都提供了命令行界面。对于.NET Core CLI，现在也是如此。前面介绍了用于创建应用程序的 dotnet 命令，还使用了一些扩展，如 dotnet user-secrets(第 24 章)和 dotnet ef(第 26 章)。这种体验现在更适合用于创建 Web 应用程序的工具。另一方面，Visual Studio 提供了一些 Web 工具的集成。

开发 Web 应用程序的客户端部分需要如下工具：

- 下载包的工具
- 处理编译或转换源文件(例如从 TypeScript 转换为 JavaScript) 的工具
- 分析源文件的工具
- 捆绑脚本文件的工具
- 单元测试的工具

根据所使用的模板，可以集成不同的工具来使用源代码。例如，如果使用来自 dotnet CLI 的模板创建一个 ASP.NET Core MVC Web 应用程序，这些工具就集成在项目中：

- Bower 用于下载 JavaScript 库(https://www.bower.io)。
- Bundler and Minifier 是一个来自 Mads Kristensen 的 Visual Studio 扩展，用于捆绑、缩小 JavaScript 和 CSS 文件，详见 https://github.com/madskristensen/BundlerMinifier。

当使用 Visual Studio 或 dotnet CLI 创建新的 Angular 项目时，要使用的工具和库如下：

- 下载 JavaScript 包的 npm (https://www.npmjs.com)
- 把 TypeScript 文件转换为 JavaScript 文件的 tsc(TypeScript 编译器) (https://www.typescriptlang.org)
- Jasmine，一个 JavaScript 测试框架(https://jasmine.github.io/)
- Karma，测试运行器，用于测试 JavaScript 代码(http://karma-runner.github.io/1.0/index.html)
- Chai，这是一个用于单元测试的断言库 (http://chaijs.com/)
- webpack，模块打包机，用于打包、捆绑和加载 JavaScript 库(https://webpack.js.org/)

不仅可以使用模板中的工具，还可以自定义代码和项目配置，以使用最适合自己工作方式的工具。

### 30.4.2　通过 Bower 使用客户端库

.NET 包可以从 NuGet 服务器中获得。对于.NET Core, JavaScript 库在此服务器上不再可用。JavaScript 社区使用其他服务器，更灵活地更改服务器。当 Visual Studio 2015 发布时，几乎所有的客户端 JavaScript 库都可以在 Bower 服务器上使用。这就是为什么微软在 Visual Studio 中集成了像 NuGet 这样的 Bower。那时，服务器上使用的脚本库通常可以在 npm(节点包管理器)服务器(https://www.npmjs.com)上使用。现在，服务器上使用的脚本库和客户机上使用的脚本库都可以在 npm 服务器上使用。

对于.NET 项目，NuGet 包在 csproj 项目文件中管理。当使用来自 Bower 服务器的包时，Visual Studio 项目模板 Bower Configuration File 将文件 Bower.json 添加到项目中。

在 Solution Explorer 中选择 bower 配置文件时，可以管理 bower 包，并打开包管理器，例如 NuGet 包管理器(参见图 30-6)。可以像使用 NuGet 包管理器一样，浏览和搜索包、安装特定的版本、更新包。

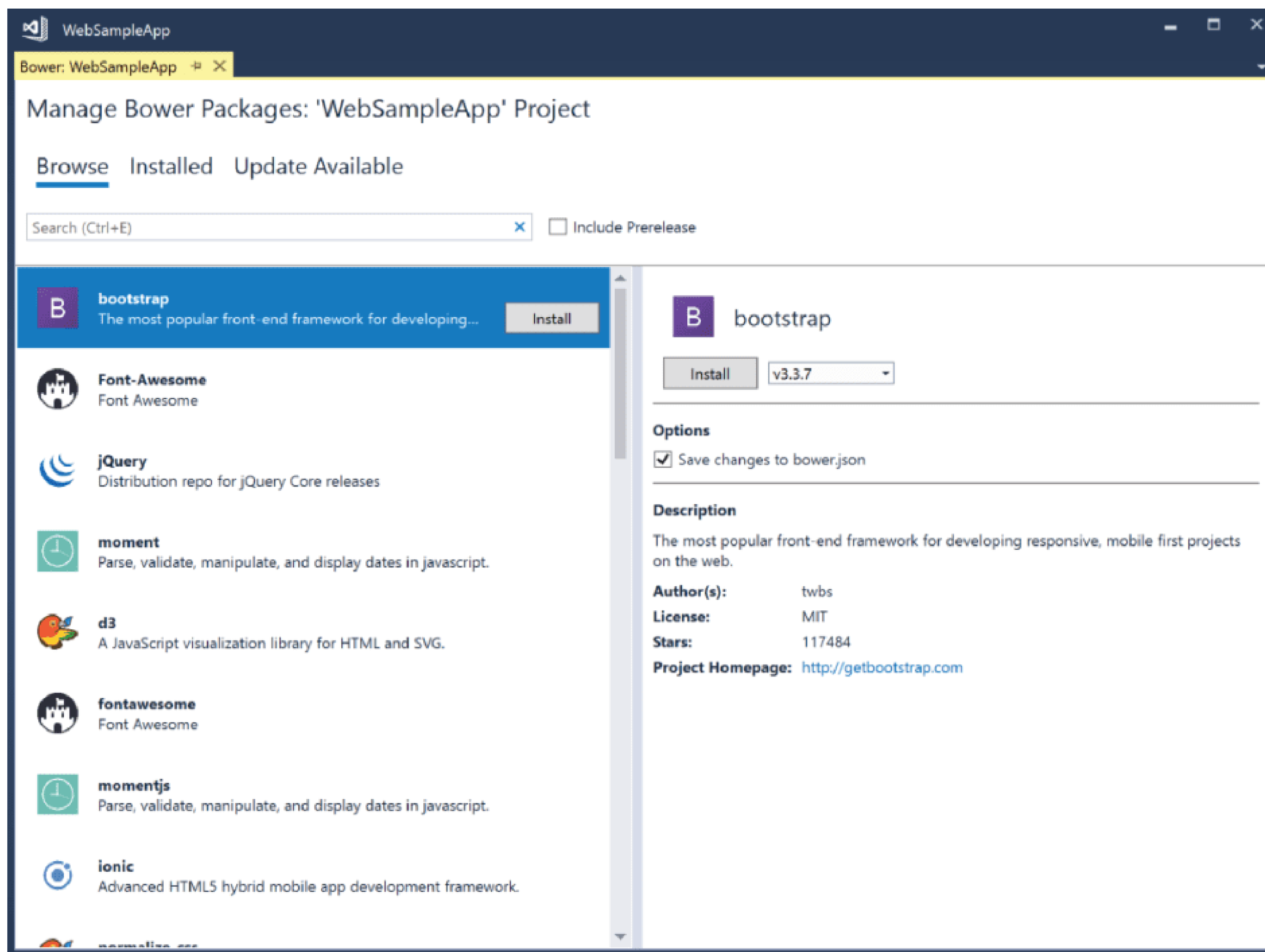



图 30-6

通过 Bower 包管理器安装 Bootstrap，会向 bower 配置文件添加一个对 bootstrap 的引用(配置文件 WebSampleApp/Bower.json)：

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "v3.3.7"
  }
}
```

在 Solution Explorer 的 Dependencies 列表中也显示了 Bower 依赖项。该包会自动下载到 wwwroot/lib。下载包的位置由.bowerrc 文件指定：

```
{
  "directory": "wwwroot/lib"
}
```

使用命令行中的 bower 时，可以使用 npm 安装 bower 命令行实用程序：

```
> npm install -g bower
```

还可以安装包，它将包写入 bower.json 文件：

```
> bower install jquery
```

### 30.4.3 使用 JavaScript 包管理器 npm

今天，Node Package Manager (npm)的主机不仅服务于服务器端 JavaScript 库，而且大多数客户端 JavaScript 库也可以从 npm 服务器中获得。

**注意：**
使用 Visual Studio 安装程序，可以将 Node Package Manager 安装为可选组件。也可以直接从 https://nodejs.org/ 上获得它。

使用 Visual Studio 2017，可以通过在项模板中添加 NPM Configuration File，将 npm 添加到项目中。添加项模板时，将下面的 package.json 文件添加到项目中：

```
{
  "version": "1.0.0",
  "name": "ASP.NET",
  "private": "true",
  "devDependencies": {
  }
}
```

devDependencies 是用来描述仅在开发过程中需要的库的部分。dependencies 部分用于在运行期间所需的库；这些需要部署到生产服务器上。

可以将库及其版本添加到 package.json 中相应的部分。Visual Studio 提供智能感知，并与包服务器联系，以获取包名称和可用版本。或者，可以使用 npm 命令行来添加包——例如，如下面的命令行语句所示，其中的选项是--save 将依赖项写入 package.json 文件：

```
> npm install @angular/core --save
```

在 Visual Studio 编辑器中选择版本号时，可以选择^与 ~前缀。如果没有前缀，则从服务器中检索具有用户输入的确切名称的库版本。有了^前缀，就检索与主版本号相同的最新库；有了~前缀，就检索与次版本号相同的最新库。

在添加包之后，可以在 Solution Explorer 的 Dependencies 部分中使用 npm 节点轻松地更新或卸载包。

### 30.4.4 捆绑

向浏览器返回 JavaScript 和 CSS 文件在生产环境中应该与在开发环境中不同。注释和空白可以删除——这称为“缩小”过程——多个文件可以合并到一个文件中——这就是所谓的“捆绑”。缩小和捆绑都提高了性能。缩小会减小文件的字节数，绑定减少了网络传输的数量。

捆绑的一个选项是 Visual Studio 2017 集成的 Bundler 和 Minifier。只需要添加 bundleconfig.json 文件(自动

从 ASP .NET Core MVC 项目模板中添加)，如下所示。该文件包含有 outputFileName 和 inputFiles 指令的部分。输入文件被缩小并绑定，以创建一个输出文件。可以将所有 CSS 文件打包到一个 CSS 文件中，将项目的 JavaScript 文件打包到一个 JavaScript 文件中：

```
[
  {
    "outputFileName": "wwwroot/css/site.min.css",
    // An array of relative input file paths. Globbing patterns supported
    "inputFiles": [
      "wwwroot/css/site.css"
    ]
  },
  {
    "outputFileName": "wwwroot/js/site.min.js",
    "inputFiles": [
      "wwwroot/js/site.js"
    ],
    // Optionally specify minification options
    "minify": {
      "enabled": true,
      "renameLocals": true
    },
    // Optionally generate .map file
    "sourceMap": false
  }
]
```

## 30.4.5 用 webpack 打包

如前所述，webpack 是一种打包 Web 项目的现代方式(撰写本文时)。在用于 Angular 的 ASP.NET Core 模板中使用 webpack，创建文件 webpack.config.js，为 Web 应用程序配置 webpack。在这个文件中，可以找到绑定 JavaScript 和 CSS 文件的捆绑配置。

要启动 webpack，还要检查.NET 项目文件 csproj。对于 Angular 项目 AngularWithDotnetCore，这个文件包含了在构建之前运行的 DebugRunWebpack 任务，并执行 JavaScript 文件 webpack：

```
<Target Name="DebugRunWebpack" BeforeTargets="Build"
  Condition=" '$(Configuration)' == 'Debug' And !Exists('wwwroot\dist') ">
  <!-- Ensure Node.js is installed -->
  <Exec Command="node --version" ContinueOnError="true">
    <Output TaskParameter="ExitCode" PropertyName="ErrorCode" />
  </Exec>
  <Error Condition="'$(ErrorCode)' != '0'"
    Text="Node.js is required to build and run this project. To continue,
please install Node.js from https://nodejs.org/, and then restart your command
prompt or IDE." />

  <!-- In development, the dist files won't exist on the first run or when
    cloning to a different machine, so rebuild them if not already present. -->
  <Message Importance="high" Text="Performing first-run Webpack build..." />
  <Exec Command="node node_modules/webpack/bin/webpack.js
    --config webpack.config.vendor.js" />
  <Exec Command="node node_modules/webpack/bin/webpack.js" />
</Target>
```

使用任务 PublishRunWebpack，其中安装了 npm 模块，webpack 从缩小和捆绑开始：

```
<Target Name="PublishRunWebpack" AfterTargets="ComputeFilesToPublish">
  <!-- As part of publishing, ensure the JS resources are freshly built in
    production mode -->
  <Exec Command="npm install" />
  <Exec Command="node node_modules/webpack/bin/webpack.js --config
    webpack.config.vendor.js --env.prod" />
  <Exec Command="node node_modules/webpack/bin/webpack.js --env.prod" />

  <!-- Include the newly-built files in the publish output -->
  <ItemGroup>
    <DistFiles Include="wwwroot\dist\**; ClientApp\dist\**" />
    <ResolvedFileToPublish Include="@(DistFiles->'%(FullPath)')"
      Exclude="@(ResolvedFileToPublish)">
      <RelativePath>%(DistFiles.Identity)</RelativePath>
      <CopyToPublishDirectory>PreserveNewest</CopyToPublishDirectory>
```

```
    </ResolvedFileToPublish>
  </ItemGroup>
</Target>
```

# 30.5　请求和响应

客户端通过 HTTP 协议向服务器发出请求。这个请求用 HTTP 响应来回答。

请求包含发送给服务器的标题和(在许多情况下)请求体信息。服务器使用标题信息了解客户端的需求，基于这个信息发送不同的结果。下面看看可以从客户端读取的信息。

为了把 HTML 格式的输出返回到客户端，span 和 Div 方法会创建一个 HTML div 元素，其中包含 HTML span 元素与传递的参数 key 和 value(代码文件 WebSampleApp/HtmlExtensions)：

```
public static string Span(this string value) =>
  $"<span>{value}</span>";

public static string Div(this string key, string value) =>
  $"{key.Span()}: {value.Span()}".Div();
```

GetRequestInformation 方法使用 HttpRequest 对象访问 Scheme、Host、Path、QueryString、Method 和 Protocol 属性(代码文件 WebSampleApp/RequestAndResponseSamples.cs)：

```
public static string GetRequestInformation(HttpRequest request)
{
  var sb = new StringBuilder();
  sb.Append("scheme".Div(request.Scheme));
  sb.Append("host".Div(request.Host.HasValue ? request.Host.Value :
    "no host"));
  sb.Append("path".Div(request.Path));
  sb.Append("query string".Div(request.QueryString.HasValue ?
    request.QueryString.Value : "no query string"));
  sb.Append("method".Div(request.Method));
  sb.Append("protocol".Div(request.Protocol));
  return sb.ToString();
}
```

把路径/RequestAndResponse 传递给服务器，来处理所有用于演示本节示例代码的请求。因此 Map 方法在 Startup 类的 Configure 方法中定义：

```
app.Map("/RequestAndResponse", app1 =>
{
  app1.Run(async context =>
  {
    //...
  }
}
```

**注意：**
使用 Map 方法的路由详见本章后面的 30.8 节“简单的路由”。

Run 方法的实现代码调用 GetRequestInformation 方法，并通过 HttpContext 的 Request 属性传递 HttpRequest。结果写入 Response 对象(代码文件 WebSampleApp/Startup.cs)：

```
app1.Run(async context =>
{
  await context.Response.WriteAsync(
    RequestAndResponseSample.GetRequestInformation(context.Request));
});
```

启动程序，访问 http://localhost:32146/RequestAndResponse/，得到以下信息：

```
scheme: http
host: localhost:32146
path: /
query string: no query string
method: GET
protocol: HTTP/1.1
```

给请求添加一条路径，例如 http://localhost:32146/RequestAndResponse/Index，得到路径值集：

```
scheme:http
host:localhost:32146
path: /Index
query string: no query string
method: GET
protocol: HTTP/1.1
```

添加一个查询字符串，如 http://localhost:32146/RequestAndResponse/Sub?x=3&y=5，就会显示访问 QueryString 属性的查询字符串：

```
query string: ?x=3&y=5
```

在下面的代码片段中，使用 HttpRequest 的 Path 属性创建一个轻量级的自定义路由。根据客户端设定的路径，调用不同的方法(代码文件 WebSampleApp/Startup.cs)：

```
app.Run(async (context) =>
{
  context.Response.ContentType = "text/html";
  string result = string.Empty;
  switch (context.Request.Path.Value.ToLower())
  {
    case "/header":
      result = RequestAndResponseSamples.GetHeaderInformation(context.Request);
      break;
    case "/add":
      result = RequestAndResponseSamples.QueryString(context.Request);
      break;
    case "/content":
      result = RequestAndResponseSamples.Content(context.Request);
      break;
    case "/encoded":
      result = RequestAndResponseSamples.ContentEncoded(context.Request);
      break;
    case "/form":
      result = RequestAndResponseSamples.GetForm(context.Request);
      break;
    case "/writecookie":
      result = RequestAndResponseSamples.WriteCookie(context.Response);
      break;
    case "/readcookie":
      result = RequestAndResponseSamples.ReadCookie(context.Request);
      break;
    case "/json":
      result = RequestAndResponseSamples.GetJson(context.Response);
      break;
    default:
      result =
        RequestAndResponseSamples.GetRequestInformation(context.Request);
      break;
  }
  await context.Response.WriteAsync(result);
});
```

以下各节实现了不同的方法来显示请求标题、查询字符串等。

### 30.5.1　请求标题

下面看看客户端在 HTTP 标题中发送的信息。为了访问 HTTP 标题信息，HttpRequest 对象定义了 Headers 属性。它的类型是 IHeaderDictionary，包含带有标题命名的字典和一个值的字符串数组。使用这个信息，先前创建的 GetDiv 方法用于将 div 元素写入客户端(代码文件 WebSampleApp/RequestAndResponseSample.cs)：

```
public static string GetHeaderInformation(HttpRequest request)
{
  var sb = new StringBuilder();
  foreach (var header in request.Headers)
  {
    sb.Append(header.Key.Div(string.Join("; ", header.Value)));
  }
  return sb.ToString();
}
```

结果取决于所使用的浏览器。下面比较一下其中的一些结果。下面的结果来自 Windows 10 触摸设备上的 Internet Explorer 11：

```
Connection: Keep-Alive
Accept: text/html,application/xhtml+xml,image/jxr,*.*
Accept-Encoding: gzip, deflate
Accept-Language: en-Us,en;q=0.8,de-AT;q=0.5,de;q=0.3
Host: localhost:32146
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; Touch; rv:11.0)
like Gecko
MS-ASPNETCORE-TOKEN: f7fd3899-4436-40a2-b736-1118f43cbef3
X-Original-Proto: http
X-Original-For: 127.0.0.1:8639
```

Google Chrome 61.0 版本显示了下面的信息，包括 AppleWebKit、Chrome 和 Safari 的版本号：

```
Connection: Keep-Alive
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,
  image/apng,*/*;q=0.8
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9
Host: localhost:32146
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
  AppleWebKit/537.36 (KHTML, like Gecko) Chrome/62.0.3202.62 Safari/537.36
Upgrade-Insecure-Requests: 1
MS-ASPNETCORE-TOKEN: f7fd3899-4436-40a2-b736-1118f43cbef3
X-Original-Proto: http
X-Original-For: 127.0.0.1:8693
```

Microsoft Edge 显示了下面的信息，包括 AppleWebKit、Chrome、Safari 和 Edge 的版本号：

```
Connection: Keep-Alive
Accept: text/html, application/xhtml+xml, image/jxr, */*
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.8,de-AT;q=0.5,de;q=0.3
Cookie: color=red
Host: localhost:32146
Referer: http://localhost:32146/
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
  AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110
  Safari/537.36 Edge/16.16299
MS-ASPNETCORE-TOKEN: f7fd3899-4436-40a2-b736-1118f43cbef3
X-Original-Proto: http
X-Original-For: 127.0.0.1:8639
```

可以从这个标题信息获得什么？

Connection 标题是 HTTP 1.1 协议的一个增强。有了这个标题，客户端可以请求保持打开连接。客户端通常使用 HTML 发出多个请求，例如获得图像、CSS 和 JavaScript 文件。服务器可能会处理请求，如果负载过高，就忽略请求，关闭连接。

Accept 标题定义了浏览器接受的 mime 格式。列表按首选格式排序。根据这些信息，可能基于客户端的需求以不同的格式返回数据。IE 喜欢 HTML 格式，其次是 XHTML 和 JXR。Google Chrome 有不同的列表。它更喜欢如下格式：HTML、XHTML、XML 和 WEBP。有了这些信息，也可以定义数量。用于输出的浏览器在列表的最后都有*.*，接受返回的所有数据。

Accept-Language 标题信息显示用户配置的语言。使用这个信息，可以返回本地化信息。本地化参见第 29 章。

**注意：**

以前，服务器保存着浏览器功能的长列表。这些列表用来了解什么功能可用于哪些浏览器。为了确定浏览器，浏览器的代理字符串用于映射功能。随着时间的推移，浏览器会给出错误的信息，甚至允许用户配置应使用的浏览器名称，以得到更多的功能(因为浏览器列表通常不在服务器上更新)。在过去，IE 经常需要进行与其他浏览器不同的编程。Microsoft Edge 非常不同于 IE，与其他供应商的浏览器有更多的共同点。这就是为什么 Microsoft Edge 会在 User-Agent 字符串中显示 Mozilla、AppleWebKit、Chrome、Safari 和 Edge 的原因。最好不要用这个 User-Agent 字符串获取可用的特性列表。相反，应以编程方式检查需要的特定功能。

前面介绍的用浏览器发送的标题信息是给非常简单的网站发送的。通常情况下会有更多的细节，如 cookie、身份验证信息和自定义信息。为了查看服务器收发的所有信息，包括标题信息，可以使用浏览器的开发工具，启动一个网络会话。这样不仅会看到发送到服务器的所有请求，还会看到标题、请求体、参数、cookie 和时间信息，如图 30-7 所示。

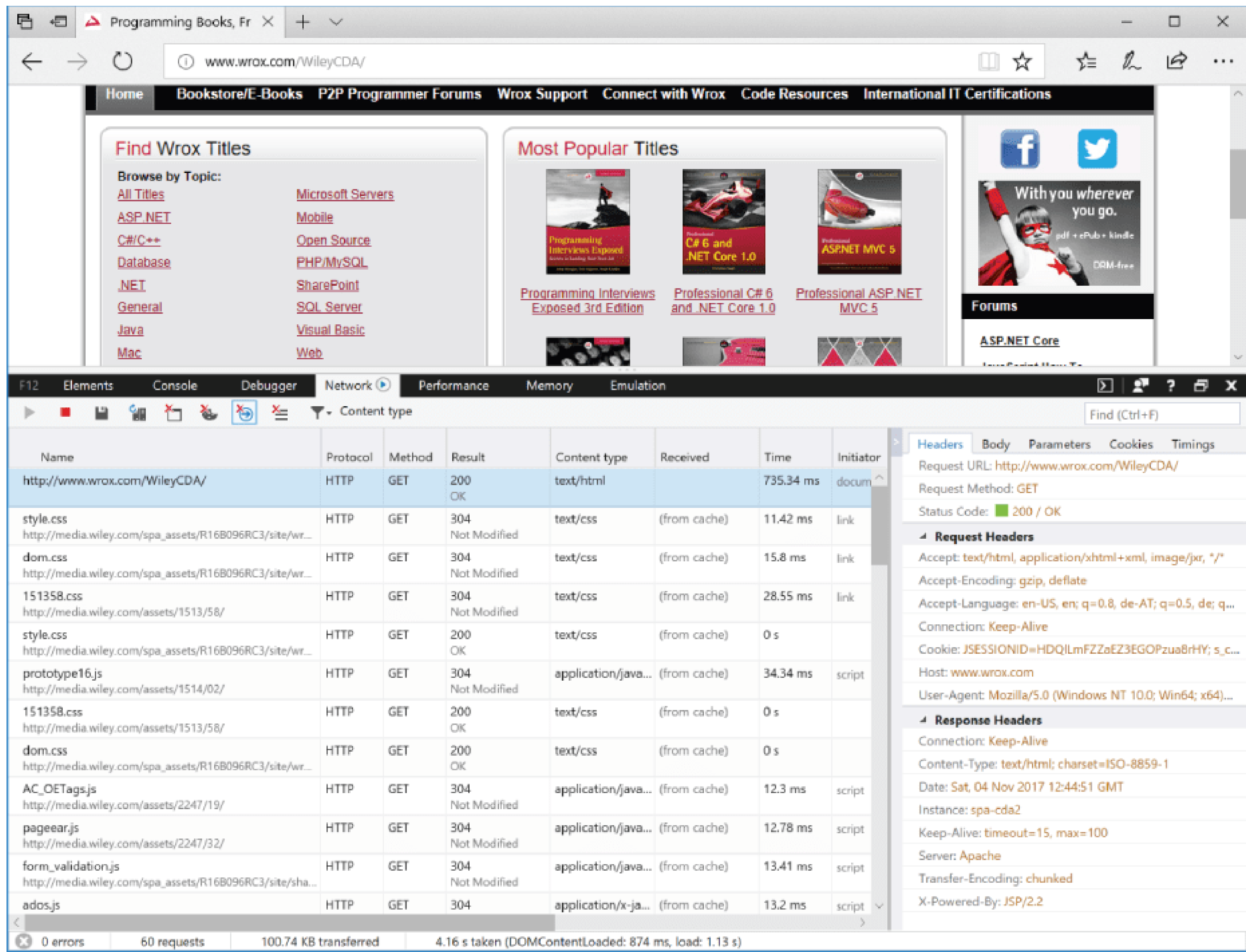



图 30-7

## 30.5.2　查询字符串

可以使用 Add 方法分析查询字符串。此方法需要 x 和 y 参数，如果这些参数是数字，就执行相加操作，并在 div 标记中返回计算结果。前一节演示的方法 GetRequestInformation 显示了如何使用 HttpRequest 对象的 QueryString 属性访问完整的查询字符串。为了访问查询字符串的各个部分，可以使用 Query 属性。下面的代码片段使用 Get 方法访问 x 和 y 的值。如果在查询字符串中没有找到相应的键，这个方法就返回 null(代码文件 WebSampleApp/RequestAndResponseSample.cs)：

```
public static string QueryString(HttpRequest request)
{
  string xtext = request.Query["x"];
  string ytext = request.Query["y"];

  if (xtext == null || ytext == null)
  {
    return "x and y must be set".Div();
  }

  if (!int.TryParse(xtext, out int x))
  {
    return $"Error parsing {xtext}".Div();
  }

  if (!int.TryParse(ytext, out int y))
  {
    return $"Error parsing {ytext}".Div();
```

```
    }
    return $"{x} + {y} = {x + y}".Div();
  }
```

从查询字符串返回的 IQueryCollection 还允许使用 Keys 属性访问所有的键，它提供了一个 ContainsKey 方法来检查指定的键是否可用。

使用 URL http://localhost:32146/RequestAndResponse/add?x=39&y=3 在浏览器中显示这个结果：

```
39 + 3 = 42
```

## 30.5.3　编码

返回用户输入的数据可能很危险。下面用 Content 方法实现这个任务。下面的方法直接返回用查询数据字符串传递的数据(代码文件 WebSampleApp/RequestAndResponseSample.cs)：

```
public static string Content(HttpRequest request) =>
  request.Query["data"];
```

使用 URL http://localhost:32146/RequestAndResponse/content?data=sample 调用这个方法，只返回字符串示例。使用相同的方法，用户还可以传递 HTML 内容，如 http://localhost:32146/RequestAndResponse/content? data =<h1> Heading 1</h1>，结果是什么？如图 30-8 显示，h1 元素由浏览器解释，文本用标题格式显示。我们有时希望这么做，例如用户(也许不是匿名用户)为一个网站写文章。

不检查用户输入，也可以让用户传递 JavaScript，如 http://localhost:32146/RequestAndResponse/content?data=<script>alert("hacker");</script>。可以使用 JavaScript 的 alert 函数弹出一个消息框。同样，很容易将用户重定向到另一个网站。当这个用户输入存储在网站中时，一个用户可以输入这样的脚本，打开这个页面的所有其他用户就会被重定向。

返回用户输入的数据应总是进行编码。下面看看不编码的结果。可以使用 HtmlEncoder 类进行 HTML 编码，如下面的代码片段所示(代码文件 WebSampleApp/RequestResponseSample.cs)：

```
public static string ContentEncoded(HttpRequest request) =>
  HtmlEncoder.Default.Encode(request.Query["data"]);
```

当应用程序运行时，进行了编码的 JavaScript 代码使用 http://localhost:32146/RequestAndResponse/encoded?data=<script>alert("hacker");</script>传递，客户就会在浏览器中看到 JavaScript 代码；它们没有被解释(参见图 30-9)。

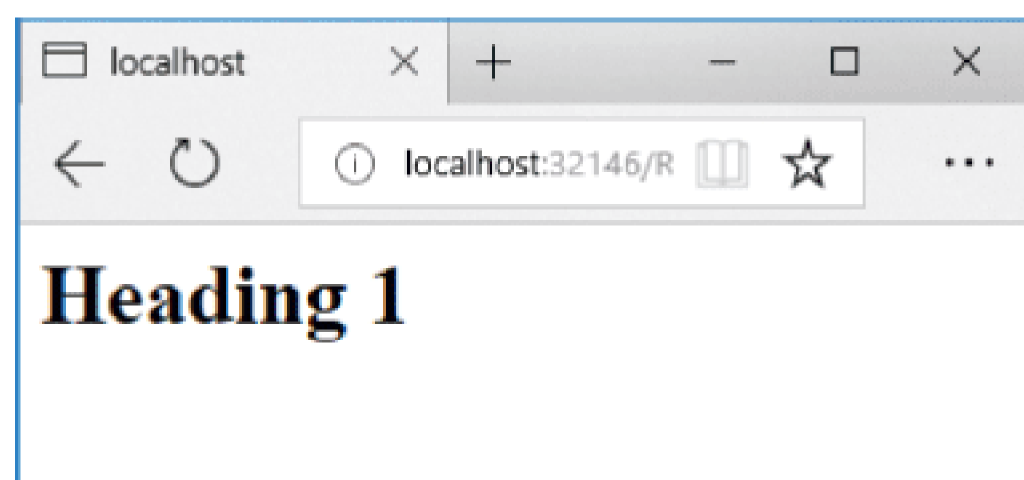



图 30-8

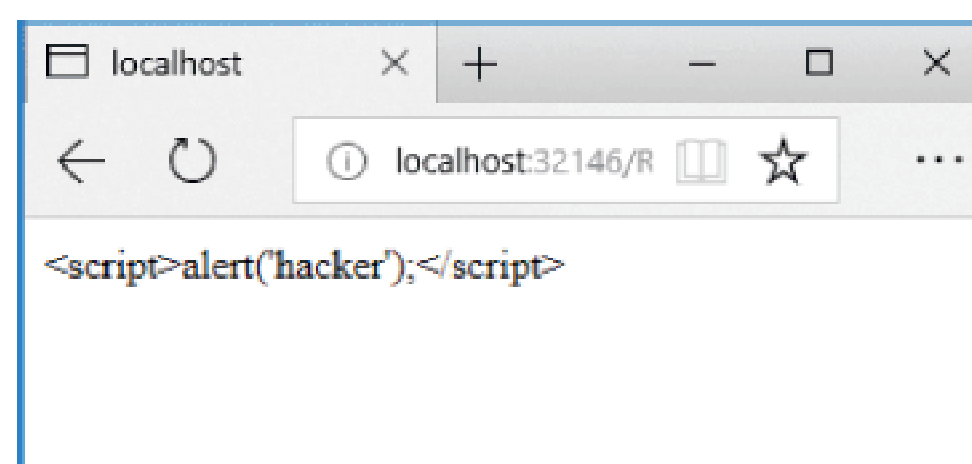



图 30-9

发送的编码字符串如下面的例子所示，有字符引用小于号(<)、大于号(>)和引号("):

```
<script>alert("hacker");</script>
```

## 30.5.4　表单数据

除了通过查询字符串把数据从用户传递给服务器之外，还可以使用表单 HTML 元素。下面这个例子使用 HTTP POST 请求替代 GET。对于 POST 请求，用户数据与请求体一起传递，而不是在查询字符串中传递。

表单数据的使用通过两个请求定义。首先，表单通过 GET 请求发送到客户端，然后用户填写表单，用 POST 请求提交数据。相应地，通过/form 路径调用的方法根据 HTTP 方法类型调用 GetForm 或 ShowForm 方法(代码文件 WebSampleApp/RequestResponseSamples.cs)：

```
public static string GetForm(HttpRequest request)
```

```
{
  string result = string.Empty;
  switch (request.Method)
  {
    case "GET":
      result = GetForm();
      break;
    case "POST":
      result = ShowForm(request);
      break;
    default:
      break;
  }
  return result;
}
```

创建一个表单，其中包含输入元素 text1 和一个 Submit 按钮。单击 Submit 按钮，调用表单的 action 方法以及用 method 参数定义的 HTTP 方法：

```
private static string GetForm() =>
  "<form method=\"post\" action=\"form\">" +
  "<input type=\"text\" name=\"text1\" />" +
  "<input type=\"submit\" value=\"Submit\" />" +
  "</form>";
```

为了读取表单数据，HttpRequest 类定义了 Form 属性。这个属性返回一个 IFormCollection 对象，其中包含发送到服务器的表单中的所有数据：

```
private static string ShowForm(HttpRequest request)
{
  var sb = new StringBuilder();
  if (request.HasFormContentType)
  {
    IFormCollection coll = request.Form;
    foreach (var key in coll.Keys)
    {
      sb.Append(key.Div(HtmlEncoder.Default.Encode(coll[key])));
    }
    return sb.ToString();
  }
  else return "no form".Div();
}
```

使用/form 链接，通过 GET 请求接收表单(参见图 30-10)。单击 Submit 按钮时，表单用 POST 请求发送，可以查看表单数据的 text1 键(参见图 30-11)。

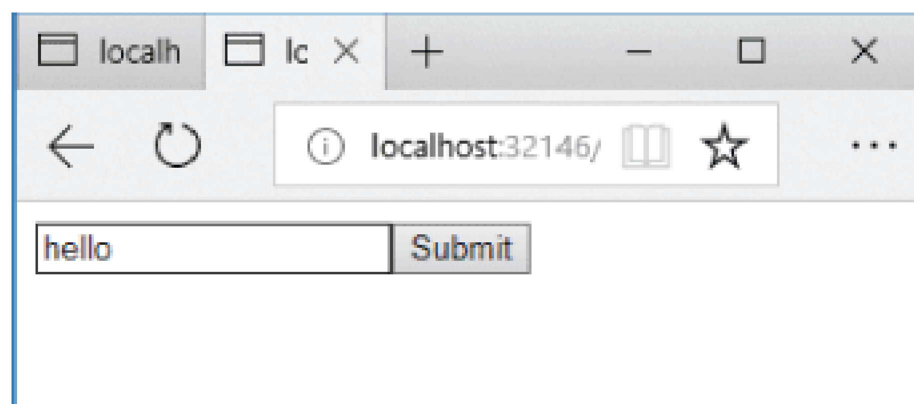



图 30-10

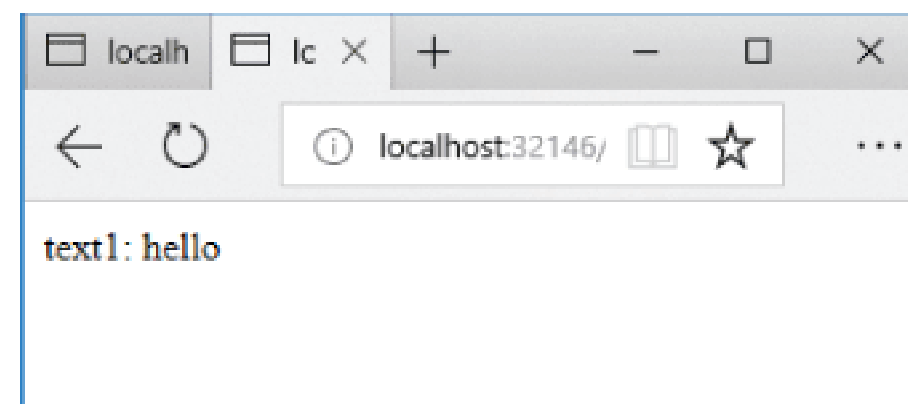



图 30-11

### 30.5.5 cookie

为了在多个请求之间记住用户数据，可以使用 cookie。给 HttpResponse 对象添加 cookie 会把 HTTP 标题内的 cookie 从服务器发送到客户端。默认情况下，cookie 是暂时的(没有存储在客户端)。如果 URL 和 cookie 在同一个域中，浏览器就将其发送回服务器。可以设置 Path 限制浏览器何时返回 cookie。在这种情况下，只有 cookie 来自同一个域且使用/cookies 路径，才返回 cookie。设置 Expires 属性时，cookie 是永久性的，因此存储在客户端。时间到了后，就删除 cookie。然而，不能保证 cookie 在之前不被删除(代码文件 WebSampleApp/Request ResponseSamples.cs)：

```
public static string WriteCookie(HttpResponse response)
{
  response.Cookies.Append("color", "red", new CookieOptions
  {
    Path = "/cookies",
    Expires = DateTime.Now.AddDays(1)
  });
```

```
  return "cookie written".Div();
}
```

通过读取 HttpRequest 对象，可以再次读取 cookie。Cookies 属性包含浏览器返回的所有 cookie：

```
public static string ReadCookie(HttpRequest request)
{
  var sb = new StringBuilder();
  IRequestCookieCollection cookies = request.Cookies;
  foreach (var key in cookies.Keys)
  {
    sb.Append(key.Div(cookies[key]));
  }
  return sb.ToString();
}
```

为了测试 cookie，还可以使用浏览器的开发工具。这些工具会显示收发的 cookie 的所有信息。

### 30.5.6 发送 JSON

服务器不仅返回 HTML 代码，还返回许多不同的数据格式，例如 CSS 文件、图像和视频。客户端通过响应标题中的 mime 类型，确定接收什么类型的数据。

GetJson 方法通过一个匿名对象创建 JSON 字符串，包括 Title、Publisher 和 Author 属性。为了用 JSON 序列化该对象，添加 NuGet 包 NewtonSoft.Json，导入 NewtonSoft.Json 名称空间。JSON 格式的 mime 类型是 application/json。这通过 HttpResponse 的 ContentType 属性来设置(代码文件 WebSampleApp/RequestResponseSample.cs)：

```
public static string GetJson(HttpResponse response)
{
  var b = new
  {
    Title = "Professional C# 7",
    Publisher = "Wrox Press",
    Author = "Christian Nagel"
  };
  string json = JsonConvert.SerializeObject(b);
  response.ContentType = "application/json";
  return json;
}
```

**注意：**

JsonConvert 类在 NuGet 包 Newtonsoft.Json 中。这个第三方包会自动从 Microsoft.AspNetCore.All 引用包中引用。

下面是返回给客户端的数据：

```
{"Title":"Professional C# 7","Publisher":"Wrox Press",
  "Author":"Christian Nagel"}
```

**注意：**

发送和接收 JSON 的内容参见第 32 章。

## 30.6 依赖注入

依赖注入深度集成在 ASP.NET Core 中。这种设计模式提供了松散耦合，因为一个服务只用一个接口。实现接口的具体类型是注入的。在 ASP.NET Core 内置的依赖注入机制中，注入通过构造函数来实现，构造函数的参数是注入的接口类型。

**注意：**

依赖注入参见第 20 章。

依赖注入将服务协定和服务实现分隔开。使用该服务时，不需要了解具体的实现，只需要一个协定。这允许在一个地方给所有使用该服务的代码替换服务(如日志记录)。

下面创建一个定制的服务，来详细论述依赖注入。

### 30.6.1 定义服务

首先，为示例服务声明一个协定。通过接口定义协定可以把服务实现及其使用分隔开，例如为单元测试使用不同的实现(代码文件 WebSampleApp/Services/ISampleService.cs)：

```
public interface ISampleService
{
  IEnumerable<string> GetSampleStrings();
}
```

用 DefaultSampleService 类实现接口 ISampleService(代码文件 WebSampleApp/Services/DefaultSampleService.cs)：

```
public class DefaultSampleService : ISampleService
{
  private List<string> _strings = new List<string> { "one", "two", "three" };
  public IEnumerable<string> GetSampleStrings() => _strings;
}
```

### 30.6.2 注册服务

使用 AddTransient 方法(这是 IServiceCollection 的一个扩展方法，在程序集 Microsoft.Extensions.DependencyInjection.Abstractions 的名称空间 Microsoft.Extensions.DependencyInjection 中定义)，DefaultSampleService 类型被映射到 ISampleService。使用 ISampleService 接口时，实例化 DefaultSampleService 类型(代码文件 WebSampleApp/Startup.cs)：

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddTransient<ISampleService, DefaultSampleService>();
  //...
}
```

内置的依赖注入服务定义了几个生命周期选项。使用 AddTransient 方法，每次注入服务时，都会实例化新的服务。

使用 AddSingleton 方法，服务只实例化一次。每次注入都使用相同的实例：

```
services.AddSingleton<ISampleService, DefaultSampleService>();
```

AddInstance 方法需要实例化一个服务，并将实例传递给该方法。这样就定义了服务的生命周期：

```
var sampleService = new DefaultSampleService();
services.AddInstance<ISampleService>(sampleService);
```

在第 4 个选项中，服务的生命周期基于当前上下文。在 ASP.NET MVC 中，当前上下文基于 HTTP 请求。只要给同样的请求调用动作，不同的注入就使用相同的实例。对于新的请求，要创建一个新实例。为了定义基于上下文的生命周期，AddScoped 方法把服务协定映射到服务上：

```
services.AddScoped<ISampleService>();
```

### 30.6.3 注入服务

注册服务之后，就可以注入它。在 Controllers 目录中创建一个控制器类型 HomeController。内置的依赖注入框架利用构造函数注入功能；因此定义一个构造函数来接收 ISampleService 接口。Index 方法接收一个 HttpContext，可以使用它读取请求信息。在实现代码中，从服务中使用 ISampleService 获得字符串。控制器添加了一些 HTML 元素，把字符串放在列表中，设置状态代码(代码文件 WebSampleApp/Controllers/HomeController.cs)：

```
public class HomeController
{
  private readonly ISampleService _service;
  public HomeController(ISampleService service) =>
```

```
    _service = service;
  public async Task Index(HttpContext context)
  {
    var sb = new StringBuilder();
    sb.Append("<ul>");
    sb.Append(string.Join(string.Empty,
      _service.GetSampleStrings().Select(s => s.Li()).ToArray()));
      sb.Append("</ul>");
      context.Response.StatusCode = 200;
      await context.Response.WriteAsync(sb.ToString());
  }
}
```

**注意：**

这个示例控制器直接返回 HTML 代码。最好把用户界面和功能分开，通过另一个类(视图)创建 HTML 代码。对于这种分离最好使用 ASP.NET MVC 框架。这个框架参见第 31 章。

### 30.6.4　调用控制器

为了通过依赖注入实例化控制器，可使用 IServiceCollection 服务注册 HomeController 类。这次不使用接口，只需要服务类型的具体实现和 AddTransient 方法调用(代码文件 WebSampleApp/Startup.cs)：

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddTransient<ISampleService, DefaultSampleService>();
  services.AddTransient<HomeController>();
  //...
}
```

包含路由信息的 Configure 方法现在改为检查/Home 路径。如果这个表达式返回 true，就在注册的应用程序服务上调用 GetService 方法，通过依赖注入实例化 HomeController。IApplicationBuilder 接口定义了 ApplicationServices 属性，它返回一个实现了 IServiceProvider 的对象。这里，可以访问所有已注册的服务。使用这个控制器，通过传递 HttpContext 来调用 Index 方法：

```
public void Configure(IApplicationBuilder app, ILoggerFactory loggerFactory)
{
  //...
  app.Map("/Home", homeApp =>
  {
    homeApp.Run(async (context) =>
    {
      HomeController controller =
        homeApp.ApplicationServices.GetService
<HomeController>();
      await controller.Index(context);
    }
  });
  //...
}
```

图 30-12 显示用主页地址的 URL 运行应用程序时的无序列表的输出。

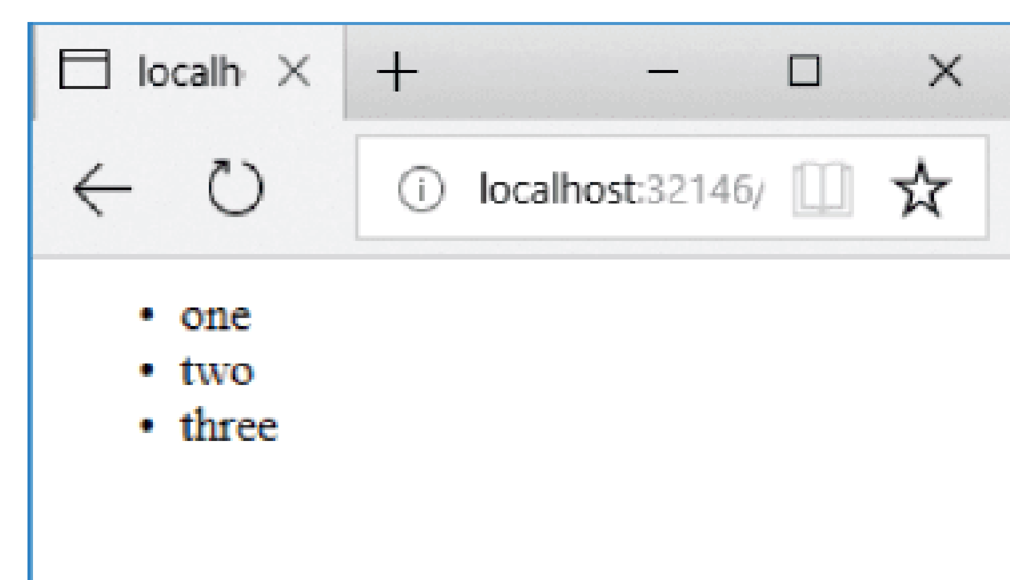



图 30-12

## 30.7　简单的路由

在前面的代码示例中使用了 Map 方法，它是 IApplicationBuilder 接口中创建简单路由的扩展方法。Map 方法通过 ASP.NET Core 提供了一个简单的路由工具。对于每个定义的映射，ASP.NET Core 提供了一个新的中间件管道，它以基于 URL 路径的 Run 方法结束。

如果收到请求，并且 Map 的路径成功，那么分配给 Action 参数的方法将定义与请求一起活动的其余管道。实现代码块中的 Run 方法指定管道的最后一步。

下面的代码段在收到请求时定义了一个到/Home 路径的映射，运行 HomeController 的 Invoke 方法(代码文

件 WebSampleApp/Startup.cs)：

```
public void Configure(IApplicationBuilder app, ILoggerFactory loggerFactory)
{
  //...
  app.Map("/Home", homeApp =>
  {
    homeApp.Run(async context =>
    {
      HomeController controller =
        app.ApplicationServices.GetService<HomeController>();
      await controller.Index(context);
    });
  });
  //...
}
```

除了使用Map方法之外，还可以使用MapWhen。在下面的代码片段中，当路径包含字符串hello时，应用MapWhen管理的映射。hello 在开头还是结尾，或者是前缀还是后缀，并不重要(代码文件 WebSampleApp/Startup.cs)：

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
  //...
  app.MapWhen(context => context.Request.Path.Value.Contains("hello"),
    helloApp =>
    {
      helloApp.Run(async context =>
      {
        await context.Response.WriteAsync("hello in the path".Div());
      });
    });
  //...
}
```

除了使用路径之外，还可以访问 HttpContext 的任何其他信息，例如客户端的主机信息(context.Request.Host)或通过身份验证的用户(context.User.Identity.IsAuthenticated)。

# 30.8　创建自定义的中间件

ASP.NET Core 很容易创建在调用 Run 方法之前调用的模块。这可以用于添加标题信息、验证令牌、构建缓存、创建日志跟踪等。一个中间件模块链接另一个中间件模块，直到调用所有连接的中间件类型为止。

使用 Visual Studio 项模板 Middleware Class 可以创建中间件类。有了这个中间件类型，就可以创建构造函数，接收对下一个中间件类型的引用。RequestDelegate 是一个委托，它接收 HttpContext 作为参数，并返回一个Task。这就是 Invoke 方法的签名。在这个方法中，可以访问请求和响应信息。HeaderMiddleware 类型给 HttpContext 的响应添加一个示例标题。在最后的动作中，Invoke 方法调用下一个中间件模块(代码文件WebSampleApp/Middleware/HeaderMiddleware.cs)：

```
public class HeaderMiddleware
{
  private readonly RequestDelegate _next;

  public HeaderMiddleware(RequestDelegate next) =>
    _next = next;

  public Task Invoke(HttpContext httpContext)
  {
    httpContext.Response.Headers.Add("sampleheader",
      new[] { "addheadermiddleware"});
    return _next(httpContext);
  }
}
```

为便于配置中间件类型，UseHeaderMiddleware 扩展方法扩展接口 IApplicationBuilder 来调用 UseMiddleware 方法：

```
public static class HeaderMiddlewareExtensions
{
  public static IApplicationBuilder UseHeaderMiddleware(
```

```
    this IApplicationBuilder builder) =>
      builder.UseMiddleware<HeaderMiddleware>();
}
```

现在，Startup 类和 Configure 方法负责配置所有的中间件类型。扩展方法已经准备好调用了(代码文件 WebSampleApp/Startup.cs)：

```
public void Configure(IApplicationBuilder app, ILoggerFactory loggerFactory)
{
  //...
  app.UseHeaderMiddleware();
  //...
}
```

运行应用程序时，可以看到返回给客户端的标题(使用浏览器的开发人员工具)。无论使用之前创建的哪个链接，每个页面都显示了标题(参见图 30-13)。

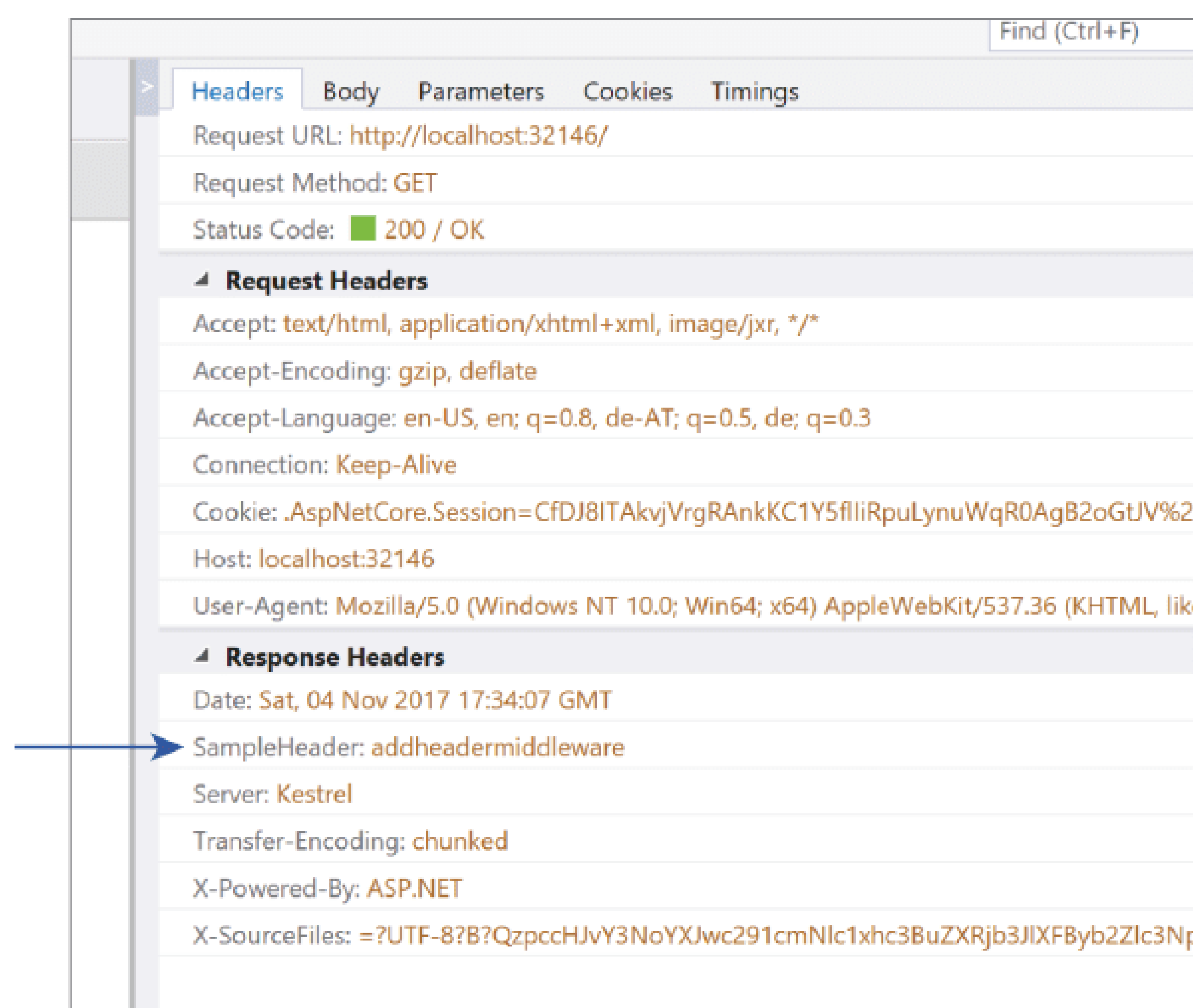



图 30-13

## 30.9　会话状态

使用中间件实现的服务是会话状态。会话状态允许在服务器上暂时记忆客户端的数据。会话状态本身实现为中间件。

用户第一次从服务器请求页面时，会启动会话状态。用户在服务器上使页面保持打开时，会话会继续到超时(通常是 10 分钟)为止。用户导航到新页面时，为了仍在服务器上保持状态，可以把状态写入一个会话。超时后，会话数据会被删除。

为了识别会话，可在第一个请求上创建一个带会话标识符的临时 cookie。这个 cookie 与每个请求一起从客户端返回到服务器，在浏览器关闭后，就删除 cookie。会话标识符也可以在 URL 字符串中发送，以替代使用 cookie。

在服务器端，会话信息可以存储在内存中。在 Web 场中，存储在内存中的会话状态不会在不同的系统之间传播。采用粘性的会话配置，用户总是返回到相同的物理服务器上。使用粘性会话，同样的状态在其他系统上不可用并不重要(除了一个服务器失败时)。不使用粘性会话，为了处理失败的服务器，应选择把会话状态存储在 SQL Server 数据库的分布式内存内。将会话状态存储在分布式内存中也有助于服务器进程的回收；如果只使用一个服务器进程，则回收处理会删除会话状态。

为了与 ASP.NET 一起使用会话状态，需要添加 NuGet 包 Microsoft.AspNet.Session。这个包提供了 AddSession 扩展方法，它可以在 Startup 类的 ConfigureServices 方法中调用。该参数允许配置闲置超时和 cookie 选项。cookie

用来识别会话。会话也使用实现了 IDistributedCache 接口的服务。一个简单的实现是进程内会话状态的缓存。方法 AddCaching 添加以下缓存服务(代码文件 WebSampleApp/Startup.cs):

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddTransient<ISampleService, DefaultSampleService>();
  services.AddTransient<HomeController>();
  services.AddDistributedMemoryCache();
  services.AddSession(options =>
    options.IdleTimeout = TimeSpan.FromMinutes(10));
}
```

为了使用会话，需要调用 UseSession 扩展方法在管道中配置中间件。在写入任何响应之前，需要调用这个方法，例如用 UseHeaderMiddleware 完成，因此 UseSession 在其他方法之前调用。使用会话信息的代码映射到/Session 路径(代码文件 WebSampleApp/Startup.cs):

```
public void Configure(IApplicationBuilder app, ILoggerFactory loggerFactory)
{
  //...
  app.UseSession();
  app.UseHeaderMiddleware();
  app.Map("/Session", sessionApp =>
  {
    sessionApp.Run(async context =>
    {
      await SessionSample.SessionAsync(context);
    });
  });
  //...
}
```

使用 Setxxx 方法可以编写会话状态，如 SetString 和 SetInt32。这些方法用 ISession 接口定义，ISession 接口从 HttpContext 的 Session 属性返回。使用 Getxxx 方法检索会话数据(代码文件 WebSampleApp/Session Sample.cs):

```
public static class SessionSample
{
  private const string SessionVisits = nameof(SessionVisits);
  private const string SessionTimeCreated = nameof(SessionTimeCreated);
  public static async Task SessionAsync(HttpContext context)
  {
    int visits = context.Session.GetInt32(SessionVisits) ?? 0;
    string timeCreated = context.Session.GetString(SessionTimeCreated) ??
      string.Empty;
    if (string.IsNullOrEmpty(timeCreated))
    {
      timeCreated = DateTime.Now.ToString("t", CultureInfo.InvariantCulture);
      context.Session.SetString(SessionTimeCreated, timeCreated);
    }
    DateTime timeCreated2 = DateTime.Parse(timeCreated);
    context.Session.SetInt32(SessionVisits, ++visits);
    await context.Response.WriteAsync(
      $"Number of visits within this session: {visits} " +
      $"that was created at {timeCreated2:T}; " +
      $"current time: {DateTime.Now:T}");
  }
}
```

**注意：**

示例代码使用不变的区域性来存储创建会话的时间。向用户显示的时间使用了特定的区域性。最好使用不变的区域性把特定区域性的数据存储在服务器上。不变的区域性和如何设置区域性参见第 26 章。

# 30.10 用 ASP.NET Core 配置

在 Web 应用程序中，需要存储可以由系统管理员改变的配置信息，例如连接字符串。下一章会创建一个数据驱动的应用程序，其中需要连接字符串。

ASP.NET Core 的配置不再像以前版本的 ASP.NET 那样基于 XML 配置文件 web.config 和 machine.config。在旧的配置文件中，程序集引用和程序集重定向是与数据库连接字符串和应用程序设置混合在一起的。现在不

再是这样。应用程序设置通常存储在 appsettings.json 中，但是配置更灵活，可以选择使用几个 JSON 或 XML 文件和环境变量进行配置。

使用 WebHost 与用默认模板创建的默认构建器时，配置了几个配置提供程序(代码文件 WebSampleApp/Program.cs)：

```
public static IWebHost BuildWebHost(string[] args) =>
  WebHost.CreateDefaultBuilder(args)
    .UseStartup<Startup>()
    .Build();
```

具体来说，这 5 个提供程序是默认配置的：

- MemoryConfigurationProvider
- JsonConfigurationProvider (appsetttings.json)
- JsonConfigurationProvider (appsettings.{environment}.json)
- EnvironmentVariablesConfigurationProvider
- CommandLineConfigurationProvider

从 Startup 类中访问配置值的一种方法是在构造函数中注入 IConfiguration 接口，并将一个属性分配给所接收的配置(代码文件 WebSampleApp/Startup.cs)：

```
public Startup(IConfiguration configuration)
{
  Configuration = configuration;
}

public IConfiguration Configuration { get; }
```

使用此内容创建应用程序配置文件(配置文件 WebSampleApp/appsettings.json)：

```
{
  "SampleSettings": {
    "Setting1": "Value1"
  },
  "AppSettings": {
    "setting2": "Value2",
    "Setting3": "Value3",
    "SubSection1": {
      "Setting4": "Value4"
    }
  },
  "ConnectionStrings": {
    "DefaultConnection":
      "Server=(localdb)\\MSSQLLocalDB;Database=_CHANGE_ME;
       Trusted_Connection=True;MultipleActiveResultSets=true"
  }
}
```

下一节将访问此设置。

### 30.10.1　读取配置

要读取配置，可以使用 IConfiguration 接口并访问各个部分。ConfigurationSample 类通过依赖注入来访问这个接口(代码文件 WebSampleApp/ConfigurationSample.cs)：

```
public class ConfigurationSample
{
  private readonly IConfiguration _configuration;
  public ConfigurationSample(IConfiguration configuration) =>
    _configuration = configuration;
  //...
}
```

可以使用索引器检索设置，可以使用 GetSection 访问配置文件中的部分。GetSection("SampleSettings")检索 SampleSettings 部分，然后访问传递字符串 Setting1 的索引器。这样，就可以检索值 Value1：

```
public async Task ShowApplicationSettingsAsync(HttpContext context)
{
  string settings = _configuration.GetSection("SampleSettings")["Setting1"];
  await context.Response.WriteAsync(settings.Div());
}
```

与使用 GetSection 方法访问层次结构配置不同，可以使用冒号语法与索引器分隔所有层次结构。下面的代码片段从配置文件中检索与前一个相同的值：

```
public async Task ShowApplicationSettingsUsingColonsAsync(HttpContext context)
{
  string settings = _configuration["SampleSettings:Setting1"];
  await context.Response.WriteAsync(settings.Div());
}
```

对于名为 ConnectionStrings 的部分，存在一个扩展方法来方便地访问连接字符串。不使用 GetSection 和索引器，而可以使用 GetConnectionString 方法，检索这个部分中的设置：

```
public async Task ShowConnectionStringSettingAsync(HttpContext context)
{
  string connectionString = _configuration
    .GetConnectionString("DefaultConnection");
  await context.Response.WriteAsync(connectionString.Div());
}
```

还可以为对配置值的强类型访问创建一个类。要使用这个特性，可以创建类 AppSettings 和 SubSection1，并将属性名直接映射到配置文件中的键(代码文件 WebSampleApp/ConfigurationSample.cs)：

```
public class SubSection1
{
  public string Setting4 { get; set; }
}

public class AppSettings
{
  public string Setting2 { get; set; }
  public string Setting3 { get; set; }
  public SubSection1 SubSection1 { get; set; }
}
```

要填充映射自定义配置类型的自定义类，应调用泛型 Get 方法，并将 AppSettings 类作为泛型参数传递。

```
public async Task ShowApplicationSettingsStronglyTyped(HttpContext context)
{
  AppSettings settings = _configuration.GetSection("AppSettings")
    .Get<AppSettings>();
  await context.Response.WriteAsync($"setting 2: {settings.Setting2}, " +
    $"setting3: {settings.Setting3}, " +
    $"setting4: {settings.SubSection1.Setting4}".Div());
}
```

对于运行应用程序并调用 ConfigurationSample 类的不同方法，MapWhen 定义一个到/Configuration 链接的映射，并将剩下的路径传递给 remainingPath 变量。根据剩下的部分，将调用来自 ConfigurationSample 类的不同方法(代码文件 WebSampleApp/Startup.cs)：

```
PathString remainingPath;
// out var not possible with lambda following in next parameter
app.MapWhen(context =>
  context.Request.Path.StartsWithSegments("/Configuration", out remainingPath),
    configurationApp =>
    {
      configurationApp.Run(async context =>
      {
        var configSample = app.ApplicationServices
          .GetService<ConfigurationSample>();
        if (remainingPath.StartsWithSegments("/appsettings"))
        {
          await configSample.ShowApplicationSettingsAsync(context);
        }
        else if (remainingPath.StartsWithSegments("/colons"))
```

```
            {
              await configSample.ShowApplicationSettingsUsingColonsAsync(context);
            }
            else if (remainingPath.StartsWithSegments("/database"))
            {
              await configSample.ShowConnectionStringSettingAsync(context);
            }
            else if (remainingPath.StartsWithSegments("/stronglytyped"))
            {
              await configSample.ShowApplicationSettingsStronglyTyped(context);
            }
          });
    });
```

### 30.10.2　修改配置提供程序

如前所述，有 5 个配置提供程序配置了默认的主机构建器。这些提供程序的顺序很重要。如果多次指定了同一键的不同值，则返回最后配置的值。例如，从命令行中传递值时，这些值是检索出来的，而不是在 JSON 文件中配置的。使用.NET Core 控制台应用程序进行完整的自定义配置，并向该应用程序添加以下 NuGet 包：

Microsoft.Extensions.Configuration

Microsoft.Extensions.Configuration.CommandLine

Microsoft.Extensions.Configuration.EnvironmentVariables

Microsoft.Extensions.Configuration.Json

控制台应用程序的配置是使用 ConfigurationBuilder 进行的。ConfigurationBuilder 类实现了接口 IConfigurationBuilder。对于这个接口，在 NuGet 包 Microsoft.Extensions.Configuration.*中定义了各种扩展方法。SetupConfiguration 方法为 JSON、环境变量和命令行添加提供程序(代码文件 CustomConfiguration/Program.cs)：

```
static void SetupConfiguration(string[] args)
{
  var builder = new ConfigurationBuilder()
    .SetBasePath(Directory.GetCurrentDirectory())
    .AddJsonFile("appsettings.json")
    .AddEnvironmentVariables()
    .AddCommandLine(args);
  Configuration = builder.Build();
}

public static IConfigurationRoot Configuration { get; private set; }
```

使用 GetSection 方法和 IConfiguration 接口的索引器读取配置，如前面几节所述：

```
private static void ReadConfiguration()
{
  string val1 = Configuration.GetSection("section1")["key1"];
  Console.WriteLine(val1);
  string val2 = Configuration.GetSection("section1")["key2"];
  Console.WriteLine(val2);
}
```

JSON 配置文件定义了这些设置(配置文件 CustomConfiguration/appsettings.json)：

```
{
  "section1": {
    "key1": "value 1",
    "key2": "value 2"
  }
}
```

因为环境变量和命令行提供程序是在 JSON 提供程序之后添加的，所以这些提供程序定义的设置会覆盖其他设置。使用带有 dotnet 命令的命令行时，应用程序的参数将在--之后分配，以将应用程序的参数与 dotnet 命令的参数分离开来。分层配置部分用冒号分隔：

```
> dotnet run -- section1:key1="settings from command line"
```

在 Debug 设置中，可以配置 Visual Studio 中的命令行参数和环境变量，如图 30-14 所示。

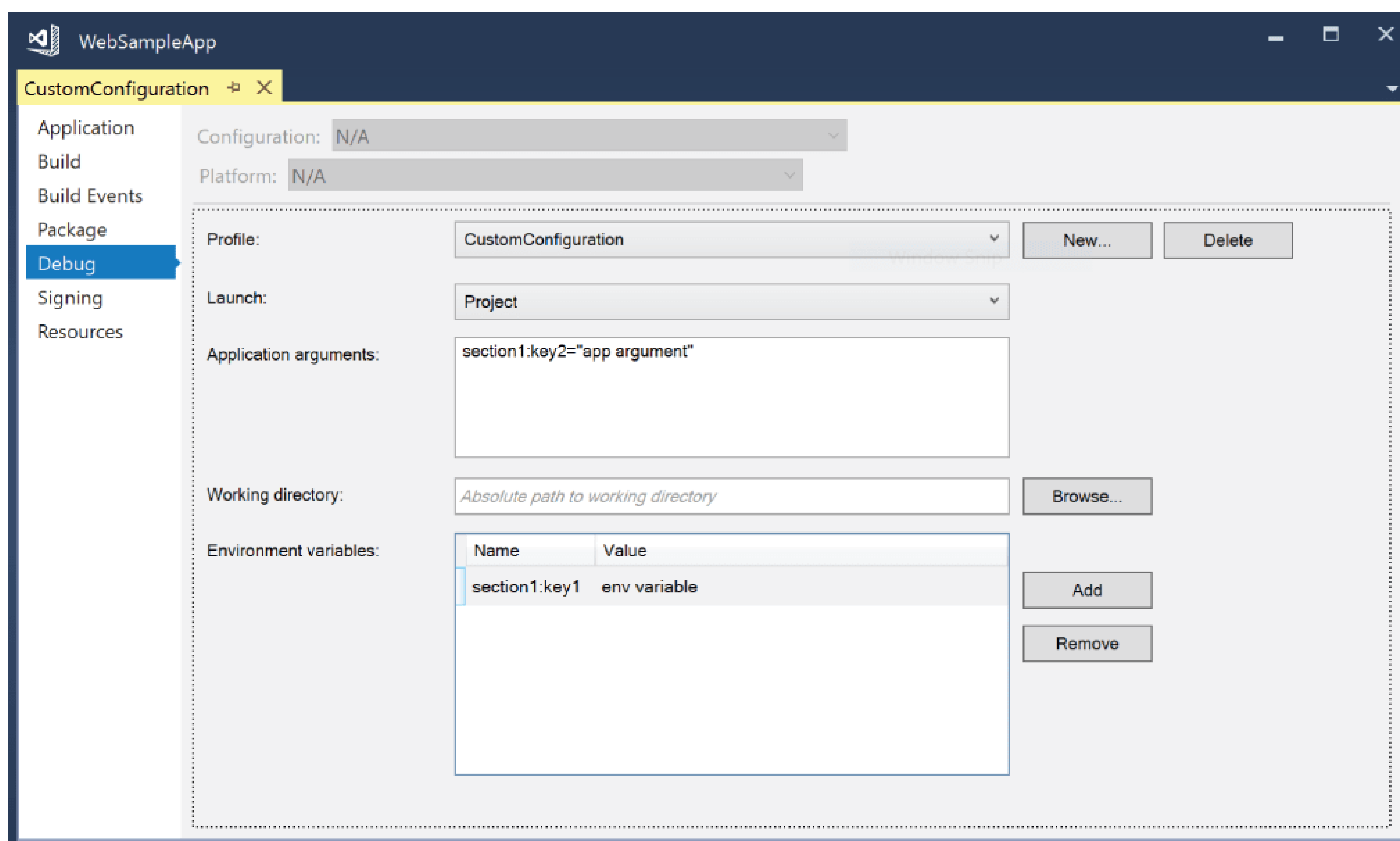



图 30-14

将调试设置写入文件 LaunchSettings. json：

```
{
  "profiles": {
    "CustomConfiguration": {
      "commandName": "Project",
      "commandLineArgs": "section1:key2=\"app argument\"",
      "environmentVariables": {
        "section1:key1": "env variable"
      }
    }
  }
}
```

在 ASP.NET Core Web 应用程序中，可以定义配置提供程序，如以前在控制台应用程序中那样。然而，ASP.NET Core 2 提供了一种更简单的方法：使用 IWebHostBuilder 的扩展方法(如 ConfigureAppConfiguration 方法)添加额外的配置。下面的代码片段使用 AddXmlFile 扩展方法添加了 XML 文件 appsettings.xml。当文件不存在时，不应该出现运行时异常，因此该文件标记为可选(代码文件 WebSampleApp/Program.cs)：

```
public static IWebHost BuildWebHost(string[] args) =>
  WebHost.CreateDefaultBuilder(args)
    .UseStartup<Startup>()
    .ConfigureAppConfiguration(configure =>
    {
      configure.AddXmlFile("appsettings.xml", optional: true);
    })
    .Build();
```

## 30.10.3　基于环境的不同配置

使用不同的环境变量运行 Web 应用程序(例如在开发、测试和生产过程中)时，也可能要使用分阶段的服务器，因为可能要使用不同的配置。测试数据不应添加到生产数据库中。对于不同的配置值，可以使用不同的配置文件。

WebHost 的默认构建器添加了两个 JSON 配置文件，appsettings.json 和 appsettings.{ env.EnvironmentName }.json。第二个文件配置为可选的，就像前一节中的 XML 文件配置一样。根据调试和发布构建，环境名称的配置

是不同的。

可以在 Visual Studio 项目属性中设置名为 ASPNETCORE_ENVIRONMENT 的环境变量，就可以配置环境，如图 30-15 所示。

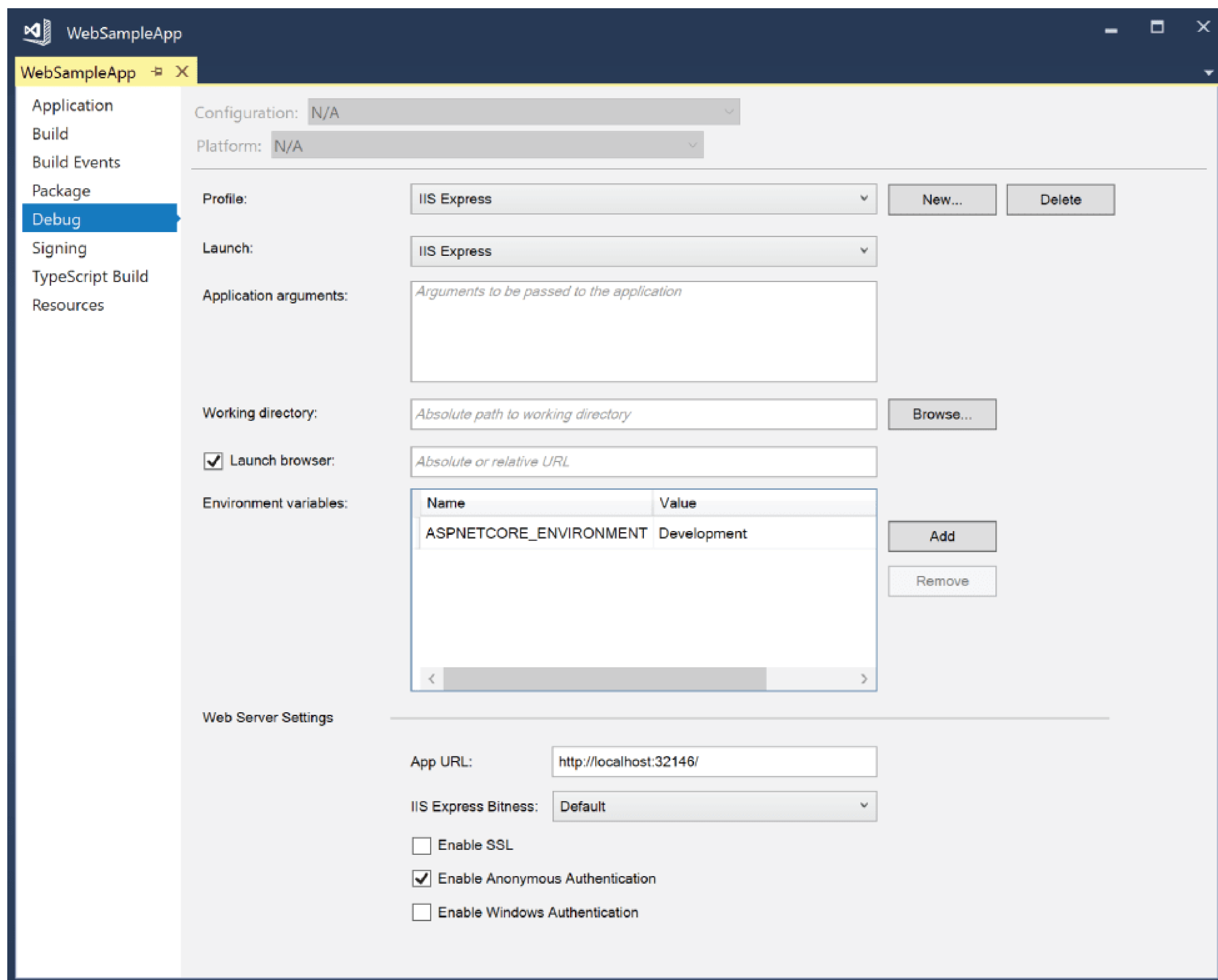



图 30-15

为了通过编程验证托管环境，可为接口 IHostingEnvironment 定义扩展方法，例如 IsDevelopment、IsStaging 和 IsProduction。为了测试任何环境名，可以给 IsEnvironment 传递验证字符串：

```
if (env.IsDevelopment())
{
  //...
}
```

# 30.11　小结

本章探讨了 ASP.NET Core 和 Web 应用程序的基础，讨论了如何处理来自浏览器的请求，并通过响应来应答。我们学习了 ASP.NET Core 依赖注射和服务的基础知识，以及使用依赖注入的具体实现，如会话状态。此外还了解了如何用不同的方式存储配置信息，例如用于不同环境(如开发和生产环境)的 JSON 配置。

第 31 章展示了 ASP.NET Core MVC 如何使用本章讨论的基础知识创建 Web 应用程序。

# 第31章 ASP.NET Core MVC

**本章要点**

- ASP.NET Core MVC 的特性
- 路由
- 创建控制器
- 创建视图
- 验证用户输入
- 使用过滤器
- 使用 HTML 和 Tag Helpers
- 创建数据驱动的 Web 应用程序
- 实现身份验证和授权
- 使用 Razor 页面

**本章源代码下载地址(wrox.com)：**

打开 www.wrox.com 的 Download Code 选项卡可下载本章源代码。源代码也可以在 MVC 目录的 https://github.com/ProfessionalCSharp/ProfessionalCSharp7 中找到。本章代码分为以下几个主要的示例文件：

- MVC Sample App
- Menu Planner
- AzureB2C Sample
- Razor Pages Sample

## 31.1 为 ASP.NET Core MVC 建立服务

第 30 章展示了 ASP.NET Core 的基础，介绍中间件以及依赖注入如何与 ASP.NET Core 一起使用。本章通过注入 ASP.NET Core MVC 服务使用依赖注入。

ASP.NET Core MVC 基于 MVC(模型-视图-控制器)模式。如图 31-1 所示，这个标准模式[*Design Patterns: Elements of Reusable Object-Oriented Software*(Addison-Wesley Professional, 1994)中记录的模式]定义了一个实现了数据实体和数据访问的模型、一个表示显示给用户的信息的视图和一个利用模型并将数据发送给视图的控制

器。控制器接收来自浏览器的请求并返回一个响应。为了建立响应，控制器可以利用模型提供一些数据，用视图定义返回的 HTML。

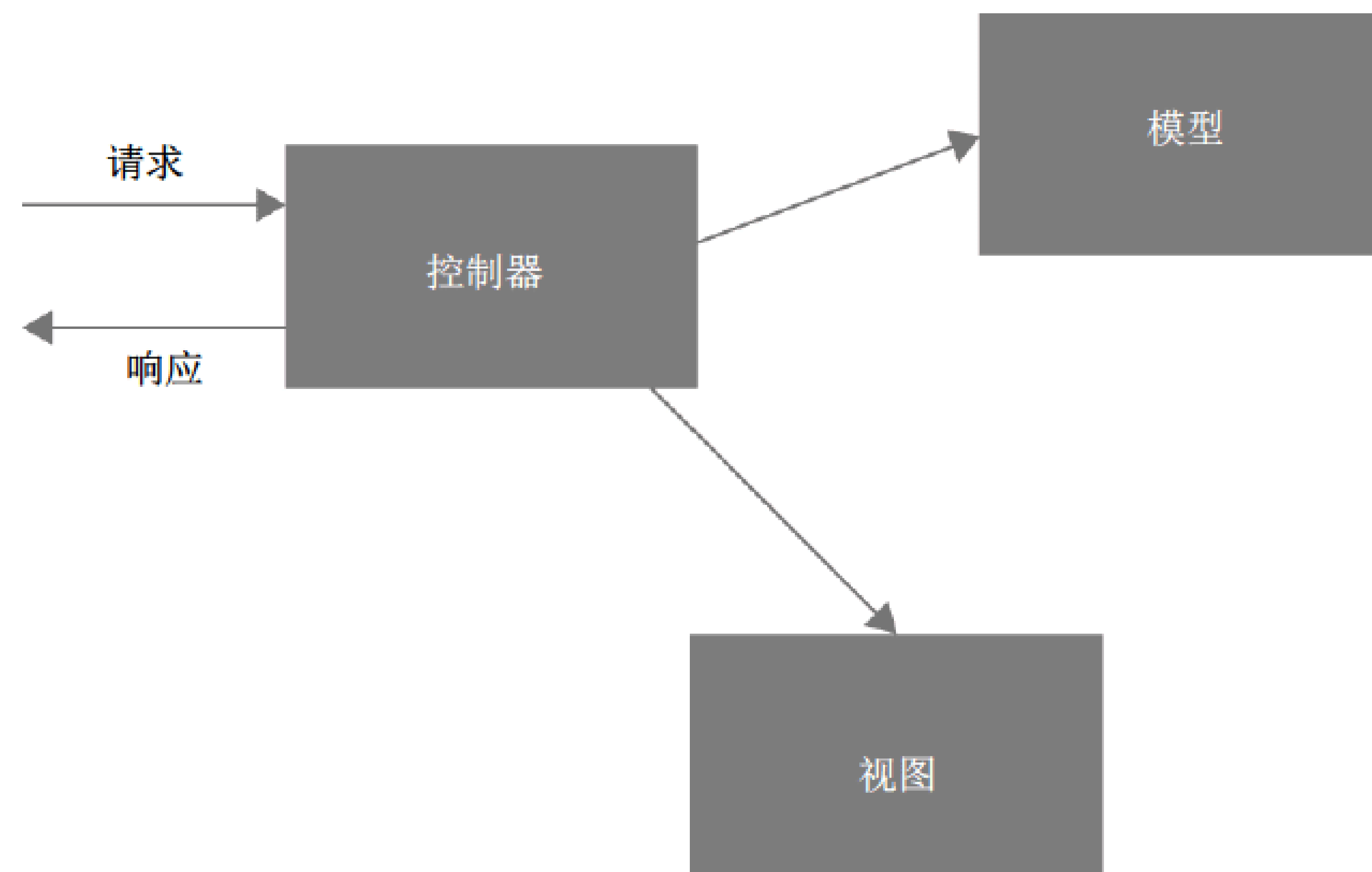



图 31-1

在 ASP.NET Core MVC 中，控制器和模型通常用服务器端运行的 C#和.NET 代码创建。视图是带有 JavaScript 的 HTML 代码，另外还有一些 C#代码用来访问服务器端信息。

这种分离在 MVC 模式中的最大好处是，可以使用单元测试方便地测试功能。控制器只包含方法，其参数和返回值可以轻松地在单元测试中覆盖。

下面开始建立 ASP.NET Core MVC 服务。在 ASP.NET Core 中，如第 30 章所述，已经深度集成了依赖注入。选择 ASP.NET Core 模板 Web Application (Model-View-Controller)可以创建一个 ASP.NET Core MVC 项目。这个模板包括 ASP.NET Core MVC 所需的 NuGet 包，以及有助于组织应用程序的目录结构。然而，这里从使用 Empty 模板开始(类似于第 30 章)，所以可以看到建立 ASP.NET Core MVC 项目都需要什么，没有项目不需要的多余东西。

创建的第一个项目命名为 MVCSampleApp。Empty 模板已经包含对 NuGet 包 Microsoft.AspNetCore.All 的引用，所以已经包含了用于 ASP.NET Core MVC 的包。有了这个包，就可在 ConfigureServices 方法中调用 AddMvc 扩展方法，添加 MVC 服务(代码文件 MVCSampleApp/Startup.cs)：

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace MVCSampleApp
{
  public class Startup
  {
    //...
    public void ConfigureServices(IServiceCollection services)
    {
      services.AddMvc();
    }
  }
}
```

AddMvc 扩展方法添加和配置几个 ASP.NET Core MVC 核心服务，如配置特性(带有 MvcOptions 和 RouteOptions 的 IConfigureOptions)；控制器工厂和控制器激活程序(IControllerFactory、IControllerActivator)；动作方法选择器、调用器和约束提供程序(IActionSelector、IActionInvokerFactory、 IActionConstraintProvider)；参数绑定器和模型验证器(IControllerActionArgumentBinder、IObjectModelValidator)以及过滤器提供程序(IFilterProvider)。

除了添加的核心服务之外，AddMvc 方法还增加了 ASP.NET Core MVC 服务来支持授权、CORS、数据注解、视图、Razor 视图引擎等。

## 31.2 定义路由

第 30 章提到，IApplicationBuilder 的 Map 扩展方法定义了一个简单的路由。本章将说明 ASP.NET Core MVC 路由基于该映射提供了一个灵活的路由机制，把 URL 映射到控制器和动作方法上。

控制器根据路由来选择。创建默认路由的一个简单方式是调用 Startup 类中的方法 UseMvcWithDefaultRoute(代码文件 MVCSampleApp/Startup.cs)：

```
public void Configure(IApplicationBuilder app)
{
  //...
  app.UseStaticFiles();
  app.UseMvcWithDefaultRoute();
  //...
}
```

> **注意：**
> *扩展方法 UseStaticFiles 参见第 30 章。*

在这个默认路由中，控制器类型的名称(没有 Controller 后缀)和方法名构成了路由，如 http://server[:port]/controller/action。也可以使用一个可选参数 id，例如 http://server[:port]/controller/action/id。控制器的默认名称是 Home；动作方法的默认名称是 Index。

下面的代码段显示了用另一种方法指定相同的默认路由。UseMvc 方法可以接收一个 Action <IRouteBuilder> 类型的参数。这个 IRouteBuilder 接口包含一个映射的路由列表。使用 MapRoute 扩展方法定义路由：

```
app.UseMvc(routes => routes.MapRoute(
  name: "default",
  template: "{controller}/{action}/{id?}",
  defaults: new {controller = "Home", action = "Index"}
));
```

这个路由定义与默认路由是一样的。template 参数定义了 URL；?与 id 一起指定这个参数是可选的；defaults 参数定义 URL 中 controller 和 action 部分的默认值。

看看下面的这个网址：

```
http://localhost:[port]/UseAService/GetSampleStrings
```

在这个 URL 中，UseAService 映射到控制器的名称，因为 Controller 后缀是自动添加的；类型名是 UseAServiceController；GetSampleStrings 是动作，代表 UseAServiceController 类型的一个方法。

### 31.2.1 添加路由

添加或修改路由的原因有几种。例如，修改路由以便使用带链接的动作、将 Home 定义为默认控制器、向链接添加额外的项或者使用多个参数。

如果要定义一个路由，让用户通过类似于 http://<server>/About 的链接来使用 Home 控制器中的 About 动作方法，而不传递控制器名称，那么可以使用如下所示的代码。URL 中省略了控制器。路由中的 controller 关键字是必须有的，但是可以定义为默认值：

```
app.UseMvc(routes => routes.MapRoute(
  name: "default",
  template: "{action}/{id?}",
  defaults: new {controller = "Home", action = "Index"}
));
```

下面显示了修改路由的另一种场景。这段代码在路由中添加了一个变量 language。该变量放在 URL 中的服务器名之后、控制器之前，如 http://server/en/Home/About。可以使用这种方法指定语言：

```
app.UseMvc(routes => routes.MapRoute(
  name: "default",
  template: "{controller}/{action}/{id?}",
  defaults: new {controller = "Home", action = "Index"}
```

```
).MapRoute(
  name: "language",
  template: "{language}/{controller}/{action}/{id?}",
  defaults: new {controller = "Home", action = "Index"}
);
```

如果一个路由匹配并找到控制器和动作方法，就使用该路由，否则选择下一个路由，直到找到匹配的路由为止。

### 31.2.2 使用路由约束

在映射路由时，可以指定约束。这样一来，就只能使用约束定义的 URL。下面的约束通过使用正则表达式(en)|(de)，定义了 language 参数只能是 en 或 de。类似于 http://<server>/en/Home/About 或 http://<server>/de/Home/About 的 URL 是合法的：

```
app.UseMvc(routes => routes.MapRoute(
  name: "language",
  template: "{language}/{controller}/{action}/{id?}",
  defaults: new {controller = "Home", action = "Index"},
  constraints: new {language = @"(en)|(de)"}
));
```

约束可以在模板定义中指定，或者通过约束定义来指定。在前面示例的 id 参数中，?后跟参数名指定，这个参数是可选的。还可以在模板中指定特定的数据类型，例如给产品 ID 使用 int 参数类型：

```
routes.MapRoute(
  name: "products",
  template: "{controller}/{action}/{productId:int}",
  defaults: new {controller = "Home", action = "Index"});
```

除了数据类型之外，还可以给最小值、最大值、范围、带有最小和最大长度的字符串指定约束。如果某个链接只允许使用数字(例如，通过产品编号访问产品)，那么可以使用正则表达式\d+来匹配多个数位构成的数字，但是至少要有一个数字：

```
app.UseMvc(routes => routes.MapRoute(
  name: "products",
  template: "{controller}/{action}/{productId?}",
  defaults: new {controller = "Home", action = "Index"},
  constraints: new {productId = @"\d+"}
));
```

**注意：**

*所有这些约束都可用，但不应给输入的验证使用约束。约束可以用于选择路由，它们本来就应该用于这种场合。如果约束不匹配，就检查下一个路由是否匹配。如果没有匹配的路由，就返回 HTTP 状态码 404(未找到)。*

路由指定了使用的控制器和控制器的动作。因此，接下来就讨论控制器。

## 31.3 创建控制器

控制器对用户请求做出反应，然后发回一个响应。如本节所述，视图并不是必要的。

ASP.NET Core MVC 中存在一些约定，优先使用约定而不是配置。对于控制器，也有一些约定。控制器位于目录 Controllers 中，控制器类的名称必须带有 Controller 后缀。

创建第一个控制器之前，先创建 Controllers 目录。然后创建一个控制器，为此要在 Solution Explorer 中选择该目录，在上下文菜单中选择 Add | New Item 命令，再选择 MVC Controller Class 项模板。对于所指定的路由，创建 HomeController。

生成的代码中包含了派生自基类 Controller 的 HomeController 类。该类中包含对应于 Index 动作的 Index 方法。请求路由定义的动作时，会调用控制器中的一个方法(代码文件 MVCSampleApp/Controllers/HomeController.cs)：

```
public class HomeController : Controller
```

```
{
  public IActionResult Index() => View();
}
```

### 31.3.1 理解动作方法

控制器中包含动作方法。下面代码段中的 Hello 方法就是一个简单的动作方法(代码文件 MVCSampleApp/Controllers/HomeController.cs)：

```
public string Hello() => "Hello, ASP.NET Core MVC";
```

使用链接 http://localhost:4994/Home/Hello 可调用 Home 控制器中的 Hello 动作。当然，端口号取决于自己的设置，可以通过项目设置中的 Web 属性进行配置。在浏览器中打开此链接后，控制器仅返回字符串 Hello, ASP.NET Core MVC。没有 HTML，而只是一个字符串。浏览器显示出了该字符串。

动作可以返回任何内容，例如图像的字节、视频、XML 或 JSON 数据，当然也可以返回 HTML。视图对于返回 HTML 很有帮助。

### 31.3.2 使用参数

如下面的代码段所示，动作方法可以声明为带有参数(代码文件 MVCSampleApp/Controllers/HomeController.cs)：

```
public string Greeting(string name) =>
  HtmlEncoder.Default.Encode($"Hello, {name}");
```

有了此声明，就可以通过请求下面的 URL 来调用 Greeting 动作方法，并在 URL http://localhost:4994/Home/Greeting?name=Stephanie 中为 name 参数传递一个值。

为了使用更易于记忆的链接，可以使用路由信息来指定参数。Greeting2 动作方法指定了参数 id：

```
public string Greeting2(string id) =>
  HtmlEncoder.Default.Encode($"Hello, {id}");
```

这匹配默认路由{controller}/{action}/{id?}，其中 id 指定为可选参数。现在可以使用此链接，id 参数包含字符串 Matthias:http://localhost:4944/Home/Greeting2/Matthias。

动作方法也可以声明为带任意数量的参数。例如，可以在 Home 控制器中添加带两个参数的 Add 动作方法：

```
public int Add(int x, int y) => x + y;
```

可以使用 URL http://localhost:4944/Home/Add?x= 4&y=5 来调用此动作，以填充 x 和 y 参数的值。

使用多个参数时，还可以定义一个路由，以在不同的链接中传递值。下面的代码段显示了路由表中定义的另一个路由，它指定了填充变量 x 和 y 的多个参数，且最多不超过 3 位数字(代码文件 MVCSampleApp/Startup.cs)：

```
app.UseMvc(routes => routes.MapRoute(
  name: "default",
  template: "{controller}/{action}/{id?}",
  defaults: new {controller = "Home", action = "Index"}
).MapRoute(
  name: "multipleparameters",
  template: "{controller}/{action}/{x:int}/{y:int}",
  defaults: new {controller = "Home", action = "Add"},
  constraints: new {x = @"\d{1,3}", y = @"\d{1,3}"}
));
```

现在可以使用 URL http://localhost:4994/Home/Add/7/2 调用与之前相同的动作。

> **注意：**
> 本章后面的 31.4.1 节“向视图传递数据”会介绍自定义类型的参数如何使用以及客户端的数据如何映射到属性上。

### 31.3.3 返回数据

到目前为止，只从控制器返回了字符串值。通常，会返回一个实现 IActionResult 接口的对象。

下面是 ResultController 类的几个例子。第一段代码使用 ContentResult 类来返回简单的文本内容。不需要创建 ContentResult 类的实例并返回该实例，而可以使用基类 Controller 的方法来返回 ActionResult。这里使用 Content 方法来返回文本内容。Content 方法允许指定内容、MIME 类型和编码(代码文件 MVCSampleApp/Controllers/ResultController.cs)：

```
public IActionResult ContentDemo() =>
  Content("Hello World", "text/plain");
```

为了返回 JSON 格式的数据，可以使用 Json 方法。下面的示例代码创建了一个 Menu 对象：

```
public IActionResult JsonDemo()
{
  var m = new Menu
  {
    Id = 3,
    Text = "Grilled sausage with sauerkraut and potatoes",
    Price = 12.90,
    Date = new DateTime(2018, 3, 31),
    Category = "Main"
  };
  return Json(m);
}
```

Menu 类定义在 Models 目录中，它定义了一个包含一些属性的简单 POCO 类(代码文件 MVCSampleApp/Models/Menu.cs)：

```
public class Menu
{
  public int Id {get; set;}
  public string Text {get; set;}
  public double Price {get; set;}
  public DateTime Date {get; set;}
  public string Category {get; set;}
}
```

客户端可在响应体内看到这些 JSON 数据，现在它们可轻松地用作 JavaScript 对象：

```
{"id":3,"text":"Grilled sausage with sauerkraut and potatoes",
"price":12.9,"date":"2018-03-31T00:00:00","category":"Main"}
```

通过使用 Controller 类的 Redirect 方法，客户端接收 HTTP 重定向请求。之后，浏览器会请求它收到的链接。Redirect 方法返回一个 RedirectResult(代码文件 MVCSampleApp/Controllers/Result Controller.cs)：

```
public IActionResult RedirectDemo() =>
  Redirect("https://www.cninnovation.com");
```

通过指定到另一个控制器和动作的重定向，也可以构建对客户端的重定向请求。RedirectToRoute 返回一个 RedirectToRouteResult，它允许指定路由名称、控制器、动作和参数。这会构建一个在收到 HTTP 重定向请求时返回客户端的链接：

```
public IActionResult RedirectRouteDemo() =>
  RedirectToRoute(new {controller = "Home", action="Hello"});
```

Controller 基类的 File 方法定义了不同的重载版本，返回不同的类型。这个方法可以返回 FileContentResult、FileStreamResult 和 VirtualFileResult。不同的返回类型取决于使用的参数，例如使用字符串返回 VirtualFileResult，使用流返回 FileStreamResult，使用字节数组返回 FileContentResult。

下一个代码段返回一幅图像。创建一个 Images 文件夹，添加一个 JPG 文件。为了让接下来的代码段执行，在 wwwroot 目录中创建一个 Images 文件夹并添加文件 Matthias.jpg。样例代码返回一个 VirtualFileResult，用第一个参数指定文件名。第二个参数用 MIME 类型 image/jpeg 指定 contentType 参数：

```
public IActionResult FileDemo() =>
  File("~/images/Matthias.jpg", "image/jpeg");
```

31.4 节“创建视图”演示了如何返回不同的 ViewResult 变体。

### 31.3.4　使用 Controller 基类和 POCO 控制器

到目前为止，创建的所有控制器都派生自基类 Controller。ASP.NET Core MVC 也支持 POCO(Plain Old CLR

Objects)控制器，它们不派生自这个基类。因此，可以使用自己的基类来定义控制器的类型层次结构。

从 Controller 基类可以得到什么？有了这个基类，控制器可以直接访问基类的属性。表 31-1 描述了这些属性和它们的功能。

表 31-1

| 属 性 | 说 明 |
| --- | --- |
| ControllerContext | 这个属性包装其他属性。在这里可以得到动作描述符的信息，其中包含动作的名称、控制器、过滤器和方法信息；直接在 HttpContext 属性中访问的 HttpContext；直接在 ModelState 属性中访问的模型状态，以及直接在 RouteData 属性中访问的路由信息 |
| HttpContext | 这个属性返回 HttpContext。在这个上下文中，可以访问 ServiceProvider 来访问依赖注入注册的服务(ApplicationServices 属性)、身份验证和用户信息、直接在 Request 和 Response 属性中访问的请求和响应信息以及 Web 套接字(如果使用它们的话) |
| ModelBinderFactory | 使用这个属性可以创建将接收到的数据绑定到动作方法的参数上的绑定器。把请求信息绑定到定制类型上的内容将在 31.5 节“从客户端提交数据”讨论 |
| MetadataProvider | 使用绑定器来绑定参数。绑定器可以利用与模型相关的元数据。使用 MetadataProvider 属性可以访问配置为处理元数据信息的提供程序的信息 |
| ModelState | ModelState 属性允许确定模型绑定是成功还是有错误。如果有错误，则可以读取导致错误的属性的信息 |
| Request | 使用这个属性可以访问 HTTP 请求的所有信息：标题和请求体信息、查询字符串、表单数据和 cookie。标题信息包含 User-Agent 字符串，它提供浏览器和客户端平台信息 |
| Response | 这个属性保存返回给客户端的信息。在这里，可以发送 cookie，改变标题信息，直接写入响应体 |
| RouteData | RouteData 属性提供了在启动代码中注册的完整路由表的信息 |
| ViewBag<br>ViewData | 使用这些属性把信息发送到视图，参见 31.4.1 节“向视图传递数据” |
| TempData | 这个属性写入在多个请求之间共享的用户状态(而可以写入 ViewBag 和 ViewData 的数据会在一个请求内的视图和控制器之间共享信息)。默认情况下，TempData 把信息写入会话状态 |
| User | User 属性返回经过身份验证的用户的信息，包括身份和声明 |

POCO 控制器没有 Controller 基类，但要访问这些信息，它仍然很重要。下面的代码段定义了一个派生自 object 基类的 POCO 控制器(可以使用自己的自定义类型作为基类)。为了用 POCO 类创建 ActionContext，可以创建一个该类型的属性。POCOController 类使用 ActionContext 作为这个属性的名称，类似于 Controller 类所采用的方式。然而，只拥有一个属性并不能自动设置它。需要应用 ActionContext 特性。使用这个特性注入实际的 ActionContext。Context 属性直接访问 ActionContext 中的 HttpContext 属性。在 UserAgentInfo 动作方法中，使用 HttpContext 属性访问和返回请求中的 User-Agent 标题信息(代码文件 MVCSampleApp/Controllers/POCOController.cs)：

```
public class POCOController
{
  public string Index() =>
    "this is a POCO controller";

  [ActionContext]
  public ActionContext ActionContext {get; set;}

  public HttpContext HttpContext => ActionContext.HttpContext;

  public ModelStateDictionary ModelState => ActionContext.ModelState;

  public string UserAgentInfo()
  {
    if (HttpContext.Request.Headers.ContainsKey("User-Agent"))
    {
      return HttpContext.Request.Headers["User-Agent"];
    }
```

```
        return "No user-agent information";
      }
    }
```

## 31.4　创建视图

返回给客户端的 HTML 代码最好通过视图指定。对于本节的示例，创建了 ViewsDemoController。视图都在 Views 文件夹中定义。ViewsDemo 控制器的视图需要一个 ViewsDemo 子目录，这是视图的约定(代码文件 MVCSampleApp/Controllers/ViewDemoController.cs)：

```
public ActionResult Index() => View();
```

> **注意：**
> 另一个可以搜索视图的地方是 Shared 目录。可以把多个控制器使用的视图(以及多个视图使用的特殊部分视图)放在 Shared 目录中。

在 Views 目录中创建 ViewsDemo 目录后，可以使用 Add | New Item 并选择 MVC View Page 项模板来创建视图。因为动作方法的名称是 Index，所以将视图文件命名为 Index.cshtml。

动作方法 Index 使用没有参数的 View 方法，因此视图引擎会在 ViewsDemo 目录中寻找与动作同名的视图文件。控制器中使用的 View 方法有重载版本，允许传递不同的视图名称。此时，视图引擎会寻找与在 View 方法中传递的名称对应的视图。

视图包含 HTML 代码，其中混合了一些服务器端代码。下面的代码段包含默认生成的 HTML 代码(代码文件 MVCSampleApp/Views/ViewsDemo/Index.cshtml)：

```
@{
  Layout = null;
}
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width" />
  <title>Index</title>
</head>
<body>
</body>
</html>
```

服务器端代码使用 Razor 语法(即有@符号)编写。31.4.2 节“Razor 语法”将讨论这种语法。在那之前，先看看如何从控制器向视图传递数据。

### 31.4.1　向视图传递数据

控制器和视图运行在同一个进程中。视图直接在控制器内创建，这便于从控制器向视图传递数据。为传递数据，可使用 ViewDataDictionary。该字典以字符串的形式存储键，并允许使用对象值。ViewDataDictionary 可以与 Controller 类的 ViewData 属性一起使用，例如向键值为 MyData 的字典传递一个字符串：ViewData["MyData"] = "Hello"。更简单的语法是使用 ViewBag 属性。ViewBag 是动态类型，允许指定任何属性名称，以向视图传递数据(代码文件 MVCSampleApp/Controllers/SubmitDataController.cs)：

```
public IActionResult PassingData()
{
  ViewBag.MyData = "Hello from the controller";
  return View();
}
```

> **注意：**
> 使用动态类型的优势在于，视图不会直接依赖于控制器。第 16 章详细介绍了动态类型。

在视图中，可以用与控制器类似的方式访问从控制器传递的数据。视图的基类 WebViewPage 定义了 ViewBag 属性(代码文件 MVCSampleApp/Views/ViewsDemo/PassingData.cshtml)：

```
<div>
  <div>@ViewBag.MyData</div>
</div>
```

### 31.4.2 Razor 语法

前面提到，视图包含 HTML 和服务器端代码。在 ASP.NET Core MVC 中，可以使用 Razor 语法在视图中编写 C#代码。Razor 使用@字符作为转换字符。@字符之后的代码是 C#代码。

使用 Razor 语法时，需要区分返回值的语句和不返回值的方法。返回的值可以直接使用。例如，ViewBag.MyData 返回一个字符串。该字符串直接放到 HTML 的 div 标记内：

```
<div>@ViewBag.MyData</div>
```

如果要调用没有返回值的方法或者指定其他不返回值的语句，则需要使用 Razor 代码块。下面的代码块定义了一个字符串变量：

```
@{
  string name = "Angela";
}
```

现在，使用转换字符@，即可通过简单的语法使用变量：

```
<div>@name</div>
```

使用 Razor 语法时，引擎在找到 HTML 元素时，会自动认为 C#代码结束。在有些情况中，这是无法自动看出来的。此时，可以使用圆括号来标记变量。其后是正常的代码：

```
<div>@(name), Stephanie</div>
```

foreach 语句也可以定义 Razor 代码块：

```
@foreach(var item in list)
{
  <li>The item name is @item.</li>
}
```

**注意：**
通常，使用 Razor 可自动检测到文本内容，例如它们以角括号开头或者使用圆括号包围变量。但在有些情况下是无法自动检测的，此时需要使用@:来显式定义文本的开始位置。

### 31.4.3 创建强类型视图

使用 ViewBag 向视图传递数据只是一种方式。另一种方式是向视图传递模型，这样可以创建强类型视图。

现在用动作方法 PassingAModel 扩展 ViewsDemoController。这里创建了 Menu 项的一个新列表，并把该列表传递给基类 Controller 的 View 方法(代码文件 MVCSampleApp/Controllers/ViewsDemoController.cs)：

```
private IEnumerable<Menu> GetSampleData() =>
  new List<Menu>
  {
    new Menu
    {
      Id=1,
      Text="Schweinsbraten mit Knödel und Sauerkraut",
      Price=6.9,
      Category="Main"
    },
    new Menu
    {
      Id=2,
      Text="Erdäpfelgulasch mit Tofu und Gebäck",
      Price=6.9,
      Category="Vegetarian"
    },
```

```
      new Menu
      {
        Id=3,
        Text="Tiroler Bauerngröst'l mit Spiegelei und Krautsalat",
        Price=6.9,
        Category="Main"
      }
    };

public IActionResult PassingAModel() =>
  View(GetSampleData());
```

当模型信息从动作方法传递到视图时，可以创建一个强类型视图。强类型视图使用 model 关键字声明。传递到视图的模型类型必须匹配 model 指令的声明。在下面的代码段中，强类型的视图声明了类型 IEnumerable<Menu>，它匹配模型类型。因为 Menu 类在名称空间 MVCSampleApp.Models 中定义，所以这个名称空间用 using 关键字打开。

通过.cshtml 文件创建的视图的基类派生自基类 RazorPage。有了模型，基类的类型就是 RazorPage <TModel>；在下面的代码段中，基类是 RazorPage<IEnumerable<Menu>>。这个泛型参数又定义了类型 IEnumerable<Menu>的 Model 属性。代码段使用基类的 Model 属性，在@foreach 中遍历 Menu 项，为每个菜单显示一个列表项(代码文件 MVCSampleApp/ViewsDemo/PassingAModel.cshtml)：

```
@using MVCSampleApp.Models
@model IEnumerable<Menu>
@{
  Layout = null;
}

<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>PassingAModel</title>
</head>
<body>
  <div>
    <ul>
      @foreach (var item in Model)
      {
        <li>@item.Text</li>
      }
    </ul>
  </div>
</body>
</html>
```

根据视图需要，可以传递任意对象作为模型。例如，编辑单个 Menu 对象时，模型的类型将是 Menu。在显示或编辑列表时，模型的类型可以是 IEnumerable<Menu>。

运行应用程序并显示定义的视图时，浏览器中将显示一个菜单列表，如图 31-2 所示。

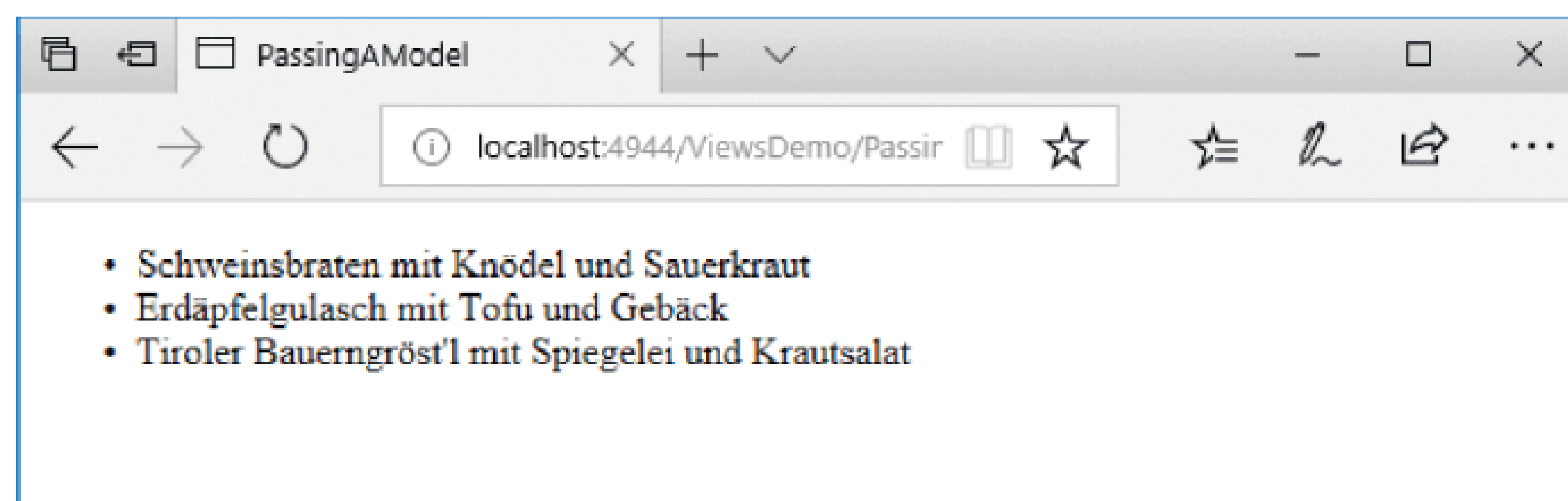



图 31-2

## 31.4.4　定义布局

通常，Web 应用程序的许多页面会显示部分相同的内容，如版权信息、徽标和主导航结构。到目前为止，所有的视图都包含完整的 HTML 内容，但有一种更简单的方式管理共享的内容，即使用布局页面。

为了定义布局，应设置视图的 Layout 属性。为了定义所有视图的默认属性，可以创建一个视图启动页面。需要把这个文件放在 Views 文件夹中，使用 MVC View Start Page 项模板创建它。这将创建_ViewStart.cshtml 文件(代码文件 MVCSampleApp/Views/_ViewStart.cshtml)：

```
@{
  Layout = "_Layout";
}
```

如果所有视图都不需要使用布局，则可以将 Layout 属性设置为 null：

```
@{
  Layout = null;
}
```

### 1. 使用默认布局页

使用 MVC View Layout Page 项模板可以创建默认的布局页面。可以在 Shared 文件夹中创建这个页面，这样它就可用于不同控制器的所有视图。Visual Studio 项模板 MVC View Layout Page 创建下面的代码：

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>@ViewBag.Title</title>
</head>
<body>
  <div>
    @RenderBody()
  </div>
</body>
</html>
```

布局页包含了所有使用该布局页的页面所共有的 HTML 内容，例如页眉、页脚和导航。前面介绍了视图和控制器如何通过 ViewBag 通信。页面布局也可以使用相同的机制。ViewBag.Title 的值可以在内容页中定义；在布局页面中，在 HTML 的 title 元素中显示它，如前面的代码段所示。基类 RazorPage 的 RenderBody 方法呈现内容页的内容，因此定义了内容应该放置的位置。

在下面的代码段中，更新生成的布局页面来引用样式表，给每个页面添加页眉、页脚和导航分区。environment、asp-controller 和 asp-action 是创建 HTML 元素的 Tag Helper。(代码文件 MVCSampleApp/Views/Shared/_Layout.cshtml)：

```
<!DOCTYPE html>
<html>
<head>
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>@ViewBag.Title</title>

  <environment include="Development">
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="~/css/site.css" />
  </environment>
  <environment exclude="Development">
    <link rel="stylesheet"
      href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css/bootstrap.min.css"
      asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
      asp-fallback-test-class="sr-only" asp-fallback-test-property="position"
      asp-fallback-test-value="absolute" />
    <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
  </environment>

  <title>@ViewBag.Title - My ASP.NET Application</title>
</head>
<body>
  <div class="container body-content">
    <header>
      <h1>ASP.NET Core MVC Sample App</h1>
    </header>
    <nav>
      <ul>
      <li><a asp-controller="ViewsDemo" asp-action="LayoutSample">
        Layout Sample</a></li>
```

```
        <li><a asp-controller="ViewsDemo" asp-action="LayoutUsingSections">
          Layout Using Sections</a></li>
        </ul>
      </nav>
      <div>
        @RenderBody()
      </div>
      <hr />
      <footer>
        <p>
          <div>Sample Code for Professional C#</div>
          &copy; @DateTime.Now.Year - My ASP.NET Application
        </p>
      </footer>
    </div>
  </body>
</html>
```

为动作 LayoutSample 创建视图(代码文件 MVCSampleApp/Views/ViewsDemo/LayoutSample.cshtml)。该视图未设置 Layout 属性，所以会使用默认布局。下面的代码设置了 ViewBag.Title，并在布局的 HTML title 元素中使用它：

```
@{
  ViewBag.Title = "Layout Sample";
}
<h2>Layout Sample</h2>
<p>
  This content is merged with the layout page
</p>
```

现在运行应用程序，布局与视图的内容会合并到一起，如图 31-3 所示。

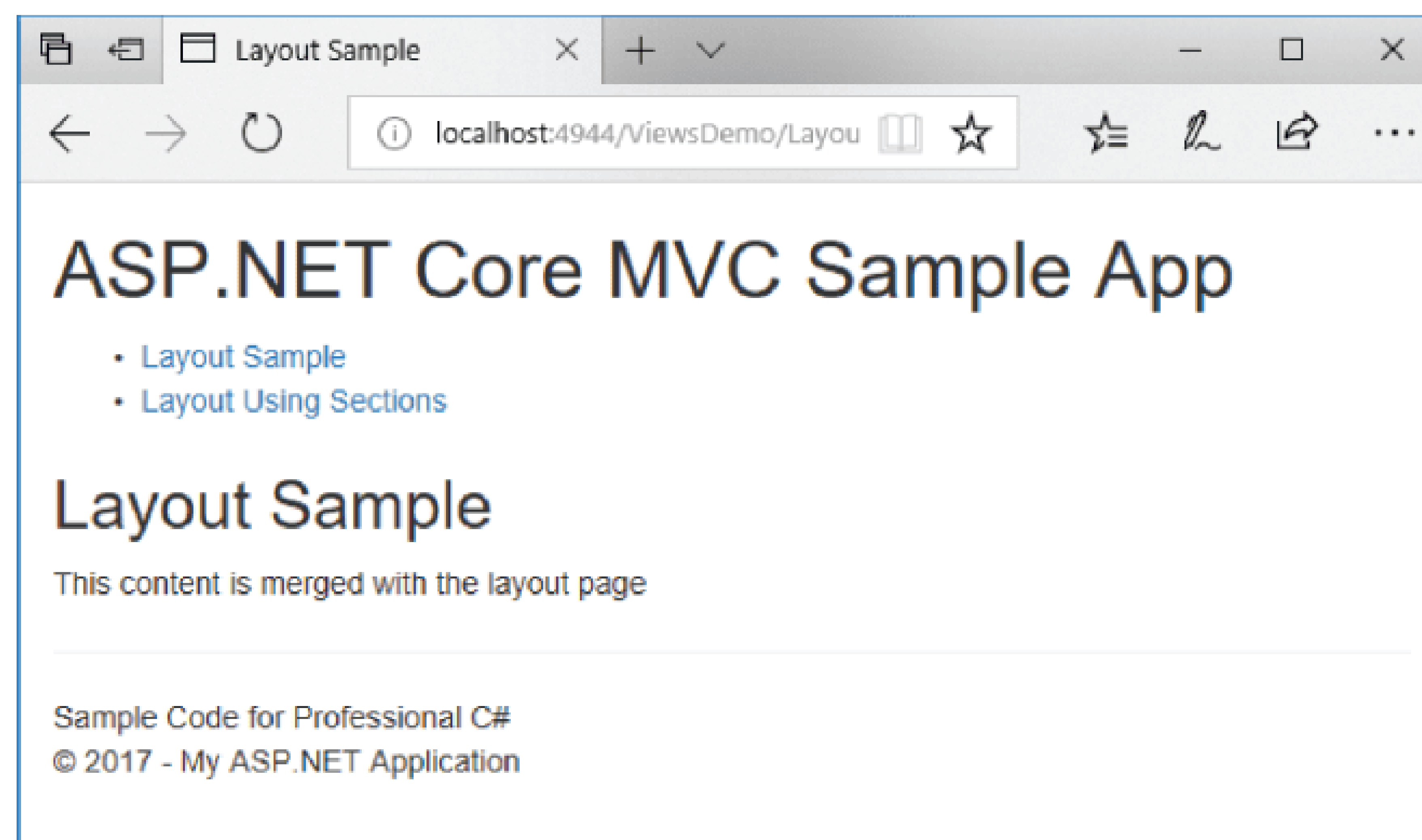



图 31-3

## 2. 使用分区

除了呈现页面主体以及使用 ViewBag 在布局和视图之间交换数据，还可以使用分区定义把视图内定义的内容放在什么位置。下面的代码段使用了一个名为 PageNavigation 的分区。默认情况下，必须有这类分区，如果没有，加载视图的操作会失败。如果把 required 参数设置为 false，该分区就变为可选(代码文件 MVCSampleApp/Views/Shared/_Layout.cshtml)：

```
<!-- ... -->
<div>
  @RenderSection("PageNavigation", required: false)
</div>
<div>
@RenderBody()
</div>
<!-- ... -->
```

在视图页面内，分区由关键字 section 定义。分区的位置与其他内容完全独立。视图没有在页面中定义位置，这是由布局定义的(代码文件 MVCSampleApp/Views/ViewsDemo/LayoutUsingSections.cshtml)：

```
@{
ViewBag.Title = "Layout Using Sections";
}
<h2>Layout Using Sections</h2>
<div>Main content here</div>
@section PageNavigation
{
  <div>Navigation defined from the view</div>
  <ul>
    <li>Nav1</li>
    <li>Nav2</li>
  </ul>
}
```

现在运行应用程序，视图与布局的内容将根据布局定义的位置合并到一起，如图 31-4 所示。

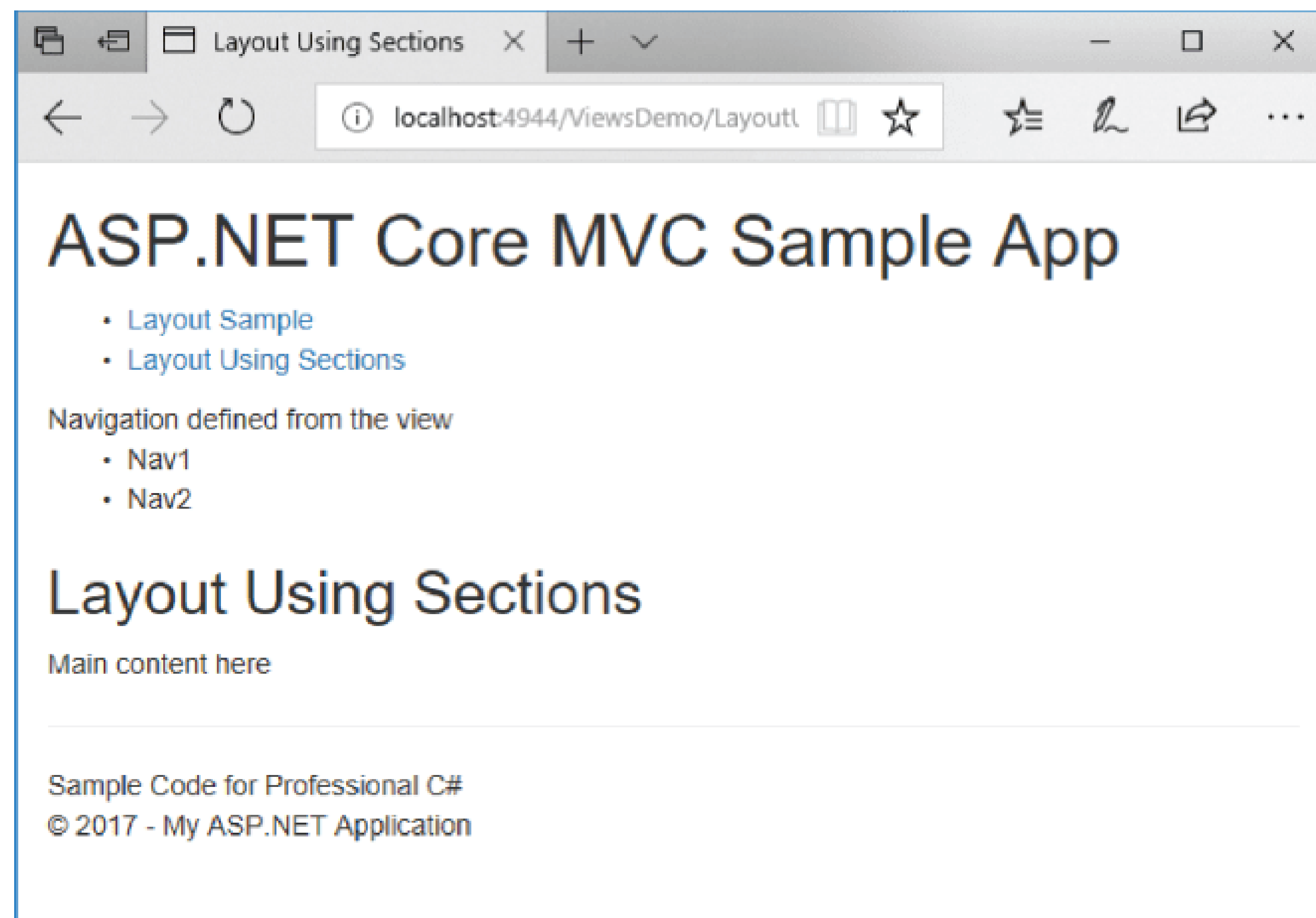



图 31-4

**注意：**

分区不只用于在 HTML 页面主体内放置一些内容，还可用于使视图在页面头部放置一些内容，如页面的元数据。

## 31.4.5 用部分视图定义内容

布局为 Web 应用程序内的多个页面提供了整体性定义，而部分视图可用于定义视图内的内容。部分视图没有布局。

此外，部分视图与标准视图类似。部分视图使用与标准视图相同的基类。

下面是部分视图的示例。首先是一个模型，它包含 EventsAndMenusContext 类定义的独立集合、事件和菜单的属性(代码文件 MVCSampleApp/Models/EventsAndMenusContext.cs)：

```
public class EventsAndMenusContext
{
  private IEnumerable<Event> GetEvents() =>
    new List<Event>
    {
      new Event(1, "Formula 1 G.P. Australia, Melbourne", new DateTime(2018, 3, 25)),
      new Event(2, "Formula 1 G.P. Bahrain, Sakhir", new DateTime(2018, 4, 8)),
      new Event(3, "Formula 1 G.P. China, Shanghai", new DateTime(2018, 4, 15)),
      new Event(4, "Formula 1 G.P. Aserbaidschan, Baku", new DateTime(2018, 4, 29))
    };

    //...

  private IEnumerable<Event> _events = null;
  public IEnumerable<Event> Events
  {
```

```
    get => _events ?? (_events = GetEvents());
  }
  private IEnumerable<Menu> _menus = null;
  public IEnumerable<Menu> Menus
  {
    get => _menus ?? (_menus = GetMenus());
  }
}
```

上下文类用依赖注入启动代码注册，通过控制器构造函数注入类型(代码文件 MVCSampleApp/Startup.cs)：

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddMvc();
  services.AddScoped<EventsAndMenusContext>();
}
```

下面使用这个模型介绍从服务器端代码加载的部分视图，然后介绍客户端的 JavaScript 代码请求的部分视图。

### 1. 使用服务器端代码中的部分视图

在 ViewsDemoController 类中，构造函数改为注入 EventsAndMenusContext 类型(代码文件 MVCSampleApp/Controllsers/ViewsDemoController.cs)：

```
public class ViewsDemoController : Controller
{
  private EventsAndMenusContext _context;
  public ViewsDemoController(EventsAndMenusContext context)
  {
    _context = context;
  }
  //...
```

动作方法 UseAPartialView1 将 EventsAndMenus 的一个实例传递给视图(代码文件 MVCSampleApp/Controllers/ViewsDemoController.cs)：

```
public IActionResult UseAPartialView1() => View(_context);
```

这个视图被定义为使用 EventsAndMenusContext 类型的模型。使用 HTML Helper Html.Par tialAsync 可以显示部分视图。该方法返回一个 Task<HtmlString>。在下面的示例代码中，使用 Razor 语法把该字符串写为 div 元素的内容。PartialAsync 方法的第一个参数接受部分视图的名称。使用第二个参数，PartialAsync 方法允许传递模型。如果没有传递模型，那么部分视图可以访问与视图相同的模型。这里，视图使用了 EventsAndMenusContext 类型的模型，部分视图只使用了该模型的一部分，所用模型的类型为 IEnumerable<Event>(代码文件 MVCSampleApp/Views/ViewsDemo/UseAPartialView1.cshtml)：

```
@using MVCSampleApp.Models
@model EventsAndMenusContext
@{
  ViewBag.Title = "Use a Partial View";
  ViewBag.EventsTitle = "Live Events";
}
<h2>Use a Partial View</h2>
<div>this is the main view</div>
<div>
  @await Html.PartialAsync("ShowEvents", Model.Events)
</div>
```

不使用异步方法的话，还可以使用同步变体 Html.Partial。这是一个扩展方法，返回实现了接口 IHtmlContent 的对象。

另外一种在视图内呈现部分视图的方法是使用 HTML Helper Html.RenderPartialAsync，该方法定义为返回 Task。该方法将部分视图的内容直接写入响应流。这样，就可以在 Razor 代码块中使用 RenderPartialAsync。

部分视图的创建方式类似于标准视图。可以访问模型，还可以使用 ViewBag 属性访问字典。部分视图会收到字典的一个副本，以接收可以使用的相同字典数据(代码文件 MVCSampleApp/Views/ViewsDemo/ShowEvents.cshtml)：

```
@using MVCSampleApp.Models
@model IEnumerable<Event>
<h2>
```

```
  @ViewBag.EventsTitle
</h2>
<table>
  @foreach (var item in Model)
  {
    <tr>
      <td>@item.Day.ToShortDateString()</td>
      <td>@item.Text</td>
    </tr>
  }
</table>
```

运行应用程序，视图、部分视图和布局都将呈现出来，如图 31-5 所示。

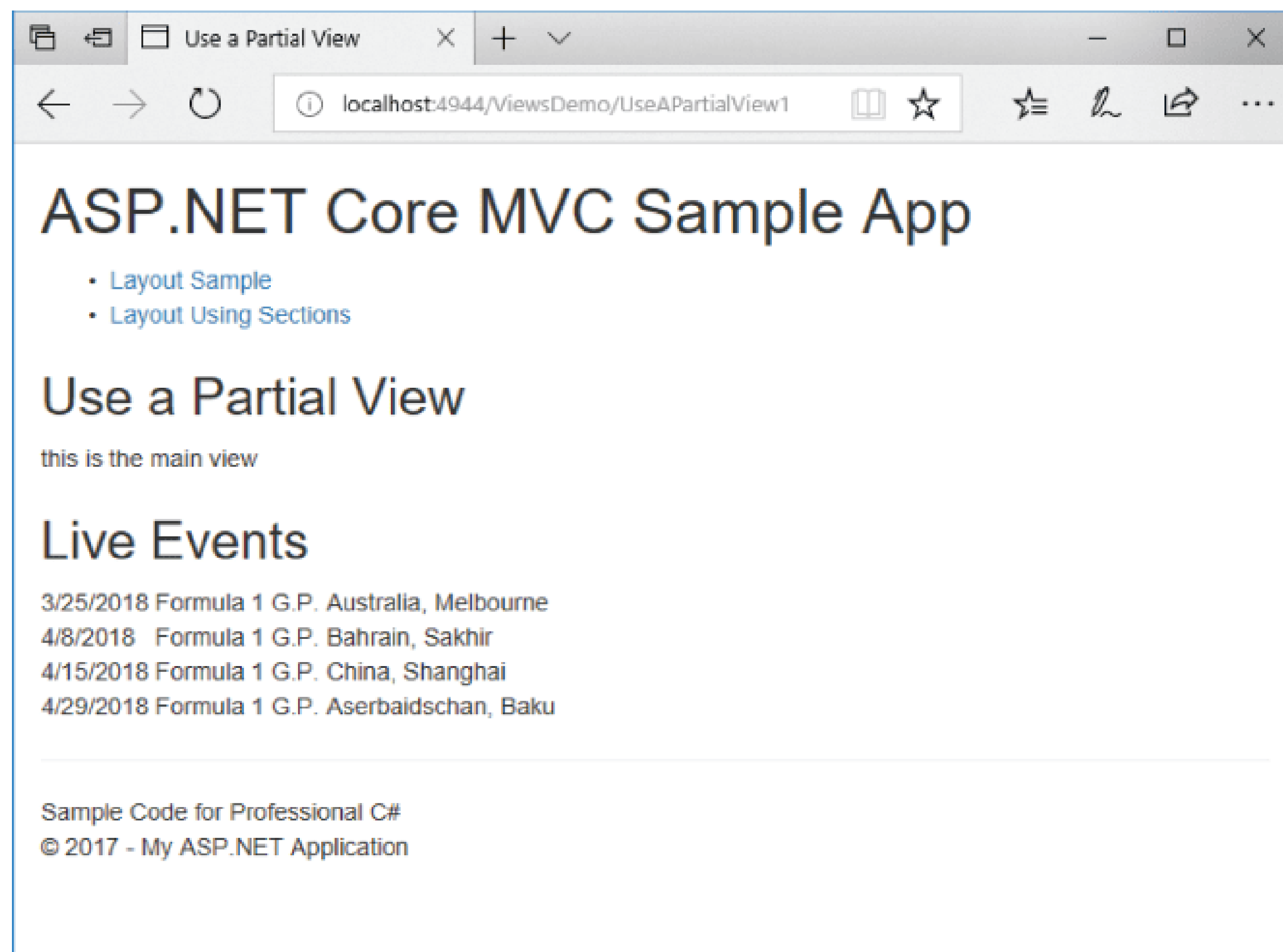



图 31-5

### 2. 从控制器中返回部分视图

到目前为止，都是直接加载部分视图，而没有与控制器交互。也可以使用控制器来返回部分视图。

在下面的代码段中，类 ViewsDemoController 内定义了两个动作方法。第一个动作方法 UsePartialView2 返回一个标准视图，第二个动作方法 ShowEvents 使用基类方法 PartialView 返回一个部分视图。前面已经创建并使用过部分视图 ShowEvents，这里再次使用它。PartialView 方法把包含事件列表的模型传递给部分视图(代码文件 MVCSampleApp/Controllers/ViewDemoController.cs)：

```
public ActionResult UseAPartialView2() => View();
public ActionResult ShowEvents()
{
  ViewBag.EventsTitle = "Live Events";
  return PartialView(_context.Events);
}
```

当部分视图在控制器中提供时，可以在客户端代码中直接调用它。下面的代码段使用了 fetch API 从服务器上下载 jQuery：事件处理程序链接到按钮的 click 事件。在单击事件处理程序内，利用 jQuery 的 load 函数向服务器发出了请求/ViewsDemo/ShowEvents 的一个 GET 请求。该请求返回一个部分视图，部分视图的结果放到了名为 events 的 div 元素内(代码文件 MVCSampleApp/Views/ViewsDemo/UseAPartial View2.cshtml)：

```
@using MVCSampleApp.Models
@model EventsAndMenusContext
@{
    ViewBag.Title = "Use a Partial View";
}
<script src="~/lib/jquery/dist/jquery.js"></script>
<script>
  (function () {
    window.onload = function () {
      var buttonEvents = document.getElementById("getEvents");
```

```
        buttonEvents.onclick = function () {
          if (window.fetch == null) {
            output.innerHTML =
              "<div>window.fetch not supported. Use another browser</div>";
            return;
          }
          var result = fetch("/ViewsDemo/ShowEvents");
          result.then(function (response) {
            return response.text();
          }).then(function (text) {
            var output = document.getElementById("output");
            output.innerHTML = text;
          });
        };
      };
    })();
</script>
<h2>Use a Partial View</h2>
<div>this is the main view</div>
<button id="getEvents">Get Events</button>
<div id="output">
</div>
```

**注意：**

fetch API 是 XmlHttpRequest API 的一个现代替代品。IE11 不支持这个 API，但 Microsoft Edge、Firefox、Safari、Chrome 和 Opera 等现代浏览器都支持它。这个新 API 详见 https://fetch.spec.whatwg.org/。

### 31.4.6　使用视图组件

ASP.NET Core MVC 提供了部分视图的新替代品：视图组件。视图组件非常类似于部分视图；主要的区别在于视图组件与控制器并不相关。这使得它很容易用于多个控制器。视图组件非常有用的例子有菜单的动态导航、登录面板或博客的侧栏内容。这些场景都独立于单个控制器。

与控制器和视图一样，视图组件也有两个部分。在视图组件中，控制器的功能由派生自 ViewComponent 的类(或带有属性 ViewComponent 的 POCO 类)接管。用户界面的定义类似于视图，但是调用视图组件的方法是不同的。

下面的代码段定义了一个派生自基类 ViewComponent 的视图组件。这个类利用前面在 Startup 类中注册的 EventsAndMenusContext 类型，可用于依赖注入。其工作原理类似于带有构造函数注入的控制器。InvokeAsync 方法定义为从显示视图组件的视图中调用。这个方法可以拥有任意数量和类型的参数，因为 IViewComponentHelper 接口定义的方法使用 params 关键字指定了数量灵活的参数。除了使用异步方法实现之外，还可以以同步方式实现该方法，返回 IViewComponentResult 而不是 Task<IViewComponentResult>。然而，通常最好使用异步变体，例如用于访问数据库。视图组件需要存储在 ViewComponents 目录中。这个目录本身可以放在项目中的任何地方(代码文件 MVCSampleApp/ViewComponents/EventListViewComponent.cs)：

```
public class EventListViewComponent : ViewComponent
{
  private readonly EventsAndMenusContext _context;
  public EventListViewComponent(EventsAndMenusContext context) =>
    _context = context;

  public Task<IViewComponentResult> InvokeAsync(DateTime from, DateTime to) =>
    Task.FromResult<IViewComponentResult>(
      View(EventsByDateRange(from, to)));

private IEnumerable<Event> EventsByDateRange(DateTime from, DateTime to) =>
  _context.Events.Where(e => e.Day >= from && e.Day <= to);
}
```

视图组件的用户界面在下面的代码段内定义。视图组件的视图可以用项模板 MVC View Page 创建；它使用相同的 Razor 语法。具体地说，它必须放入 Components/[viewcomponent]文件夹，例如 Components/EventList。为了使视图组件可用于所有的控件，需要在 Shared 文件夹中为视图创建 Components 文件夹。只使用特定控制器中的视图组件时，可以把它放到视图控制器文件夹中。这与视图的区别是，它需要命名为 default.cshtml。也

可以创建其他视图名称；但需要在 InvokeAsync 方法中使用一个参数为返回的 View 方法指定这些视图(代码文件 MVCSampleApp/Views/Shared/Components/EventList/default.cshtml)：

```
@using MVCSampleApp.Models;
@model IEnumerable<Event>
<h3>Formula 1 Calendar</h3>
<ul>
  @foreach (var ev in Model)
  {
    <li><div>@ev.Day.ToString("D")</div><div>@ev.Text</div></li>
  }
</ul>
```

现在完成了视图组件后，可以调用 InvokeAsync 方法显示它。Component 是视图的一个动态创建的属性，返回一个实现了 IViewComponentHelper 的对象。IViewComponentHelper 允许调用同步或异步方法，例如 Invoke、InvokeAsync、RenderInvoke 和 RenderInvokeAsync。当然，只能调用由视图组件实现的这些方法，参数需要用 Dictionary<string, object> 传递，其中键匹配参数名 (代码文件 MVCSampleApp/Views/ViewsDemo/UseViewComponent1.cshtml)：

```
@{
ViewBag.Title = "View Components Sample";
}
<h2>@ViewBag.Title</h2>
<p>
  @await Component.InvokeAsync("EventList", new Dictionary<string, object>()
  {
    ["from"] = new DateTime(2018, 4, 1),
    ["to"] = new DateTime(2018, 4, 20)
  })
</p>
```

自从 ASP.NET Core 1.1 以来，就可以使用 Tag Helper 调用视图组件。Tag Helper 是用 HTML 编写的——类似于简化视图的代码。创建另一个视图，来调用与以前相同的视图组件，要为视图组件使用 Tag Helper，需要添加@addTagHelper 指令和视图组件所在程序集的名称——在这个示例中是 Web 应用程序的名称。为视图组件添加 Tag Helper 之后，可以使用 vc 标记引用视图组件。在 vc 标记的冒号之后，是视图组件的名称。当类用 Pascal 命名约定定义时，Tag Helper 通过切换到小写字母来更改名称；不使用大写字母，而是添加了连字符，因此 EventList 变成 event-list(代码文件 MVCSampleApp/Views/ViewsDemo/UseViewComponent2.cshtml)：

```
@addTagHelper *, MVCSampleApp
@{
  ViewBag.Title = "View Components Sample";
}
<h2>@ViewBag.Title</h2>
<vc:event-list from="@new DateTime(2018, 4, 1)" to="@new DateTime(2018, 4, 20)" />
```

**注意：**
带小写字母和连字符的 Tag Helper 的命名规则称为 lower kebab casing。

**注意：**
Tag Helper 详见 31.7 节 “Tag Helper” 。

运行应用程序，呈现的视图组件如图 31-6 所示。

### 31.4.7　在视图中使用依赖注入

如果服务需要直接出现在视图中，可以使用 inject 关键字注入(代码文件 MVCSampleApp/Views/ViewsDemo/InjectServiceInView.cshtml)：

```
@using MVCSampleApp.Services
@inject ISampleService sampleService
<p>
  @string.Join(" * ", sampleService.GetSampleStrings())
</p>
```

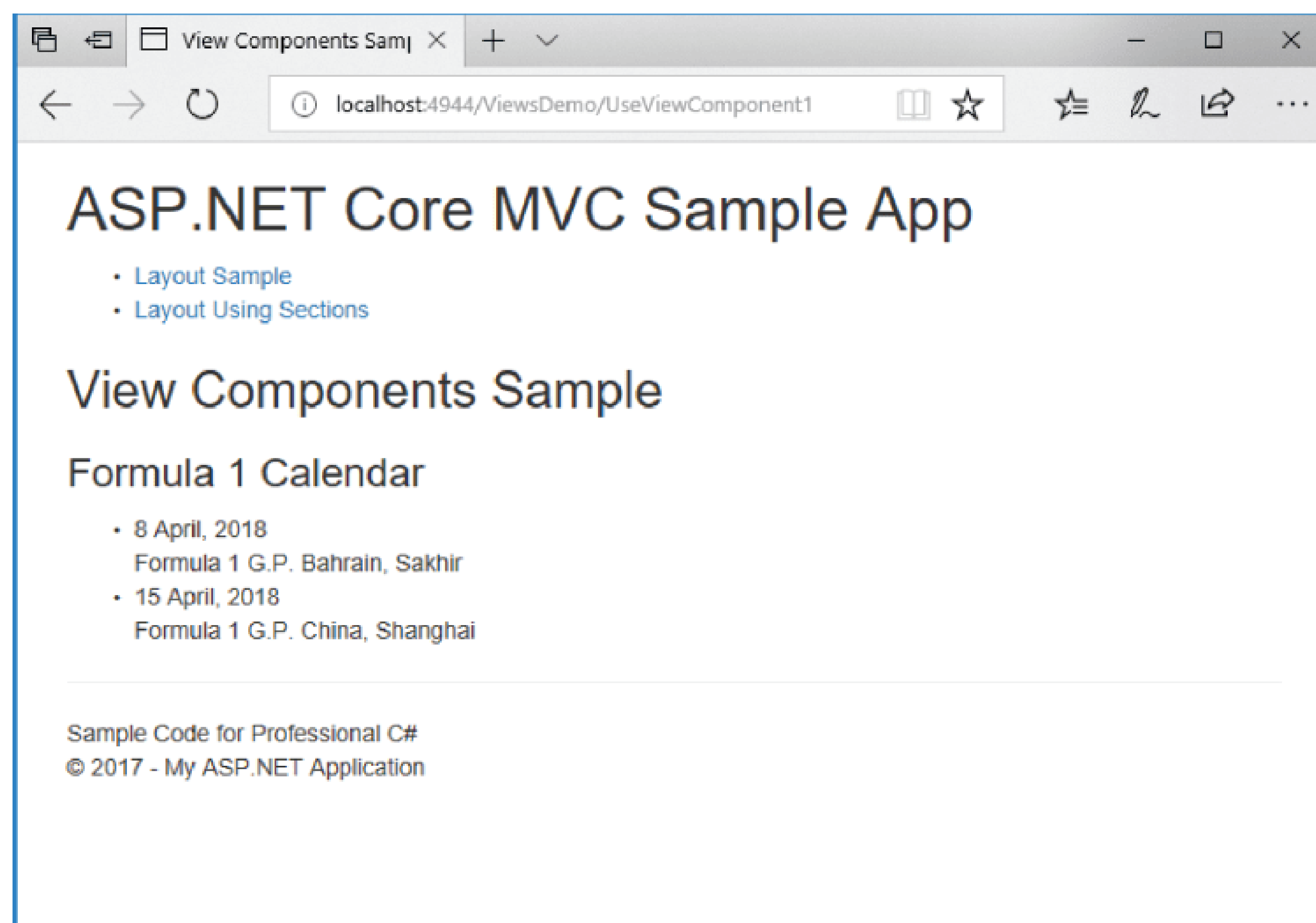



图 31-6

此时，最好使用 AddScoped 方法注册服务。如前所述，以这种方式注册服务意味着只为一个 HTTP 请求实例化一次。使用 AddScoped 在控制器和视图中注入相同的服务，也只为一个请求实例化一次。

### 31.4.8　为多个视图导入名称空间

所有之前关于视图的示例都使用 using 关键字打开了所需要的所有名称空间。除了为每个视图打开名称空间之外，还可以使用 Visual Studio 项模板 MVC View Imports Page 创建一个文件(_ViewImports.cshtml)，它定义了所有的 using 声明(代码文件 MVCSampleApp/Views/_ViewImports.cshtml)：

```
@using MVCSampleApp.Models
@using MVCSampleApp.Services
```

有了这个文件，就不需要在所有视图中添加所有的 using 关键字。

## 31.5　从客户端提交数据

到现在为止，在客户端只是使用 HTTP GET 请求来获取服务器端的 HTML 代码。那么，如何从客户端发送表单数据？

为提交表单数据，可为控制器 SubmitData 创建视图 CreateMenu。该视图包含一个 HTML 表单元素，它定义了应把什么数据发送给服务器。表单方法声明为 HTTP POST 请求。定义输入字段的 input 元素的名称全部与 Menu 类型的属性对应。对于输入元素，使用与属性类型相对应的类型。这允许在客户机上进行输入验证，而不需要编写 JavaScript 代码(代码文件 MVCSampleApp/Views/SubmitData/CreateMenu.cshtml)：

```
@{
  ViewBag.Title = "Create Menu";
}
<h2>Create Menu</h2>
<form action="/SubmitData/CreateMenu" method="post">
  <fieldset>
    <legend>Menu</legend>
    <label for="id">Id</label><br />
    <input name="id" id="id" type="number" /><br />

    <label for="text">Text</label><br />
    <input name="text" id="text" type="text" /><br />
```

```
    <label for="price">Price</label><br />
    <input name="price" id="price" type="number" step="0.01" /><br />

    <label for="date">Date</label><br />
    <input name="date" id="date" type="date" /><br />

    <label for="category">Category</label><br />
    <input name="category" id="category" type="text" /><br />

    <button type="submit">Submit</button>
  </fieldset>
</form>
```

图 31-7 显示了在浏览器中打开的页面。

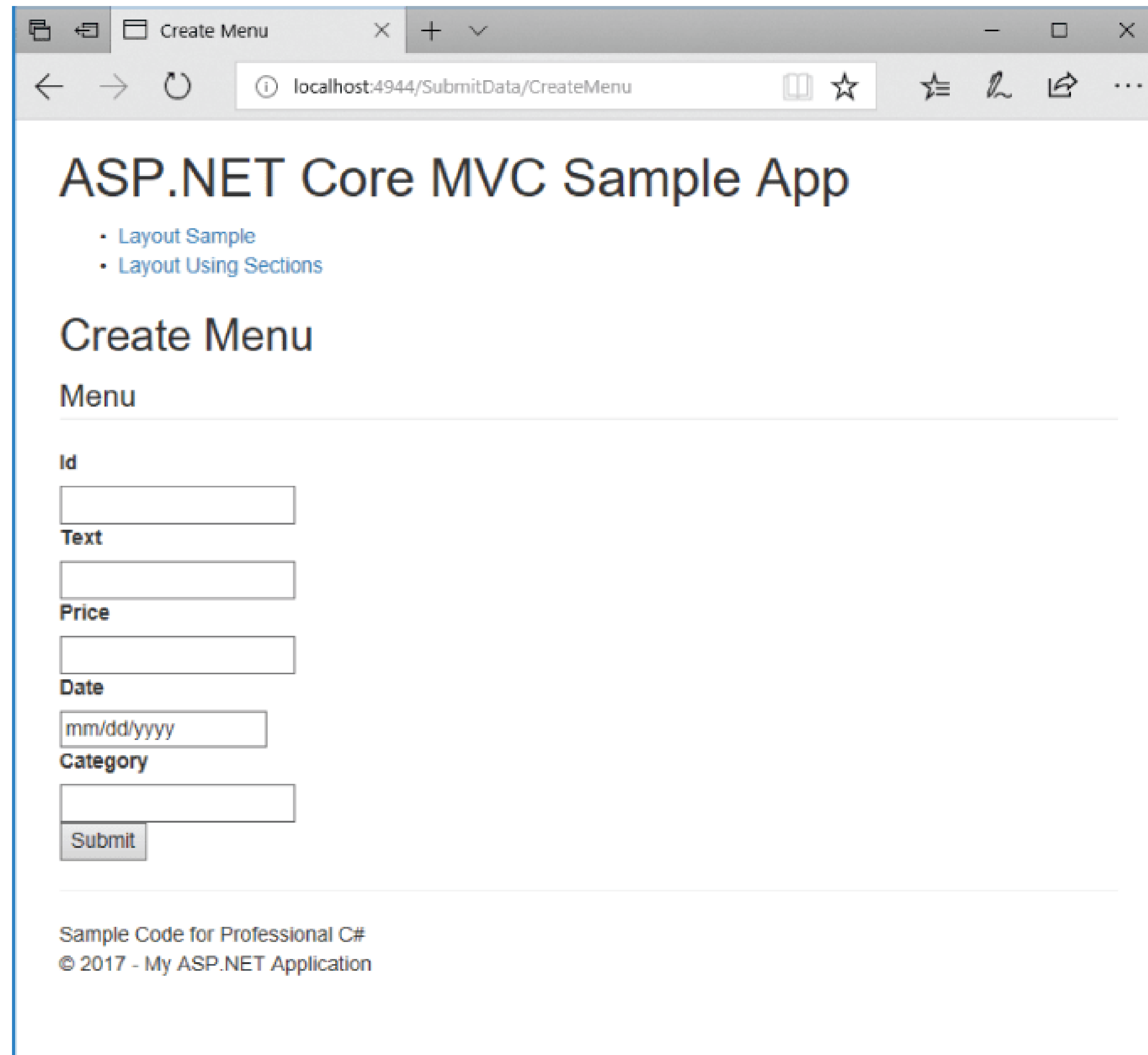



图 31-7

在 SubmitData 控制器内，创建了两个 CreateMenu 动作方法：一个用于 HTTP GET 请求，另一个用于 HTTP POST 请求。因为 C#中存在同名的不同方法，所以这些方法的参数数量或参数类型必须不同。动作方法也存在这种要求。另外，动作方法还需要与 HTTP 请求方法区分开。默认情况下，HTTP 请求方法是 GET，应用 HttpPost 特性后，请求方法是 POST。为读取 HTTP POST 数据，可以使用 Request 对象中的信息。但是，定义带参数的 CreateMenu 方法要简单多了。参数的名称与表单字段的名称匹配(代码文件 MVCSampleApp/Controllers/SubmitDataController.cs)：

```
public IActionResult Index() => View();

public IActionResult CreateMenu() => View();

[HttpPost]
public IActionResult CreateMenu(int id, string text, double price,
  DateTime date, string category)
{
  var m = new Menu
        {
          Id = id,
          Text = text,
          Price = price,
          Date = date,
          Category = category
        };
  ViewBag.Info = $"menu created: {m.Text}, Price: {m.Price}, " +
```

```
    $"date: {m.Date}, category: {m.Category}";
  return View("Index");
}
```

为了显示结果，仅显示 ViewBag.Info 的值(代码文件 MVCSampleApp/Views/SubmitData/Index.cshtml)：

```
@ViewBag.Info
```

### 31.5.1 模型绑定器

除了在动作方法中使用多个参数，还可以使用类型，类型的属性与输入的字段名称匹配(代码文件 MVCSampleApp/Controllers/SubmitDataController.cs)：

```
[HttpPost]
public IActionResult CreateMenu2(Menu menu)
{
  ViewBag.Info = $"menu created: {menu.Text}, Price: {menu.Price}, " +
    $"date: {menu.Date}, category: {menu.Category}";
  return View("Index");
}
```

提交表单数据时，会调用 CreateMenu 方法，它在 Index 视图中显示了提交的菜单数据，如图 31-8 所示。

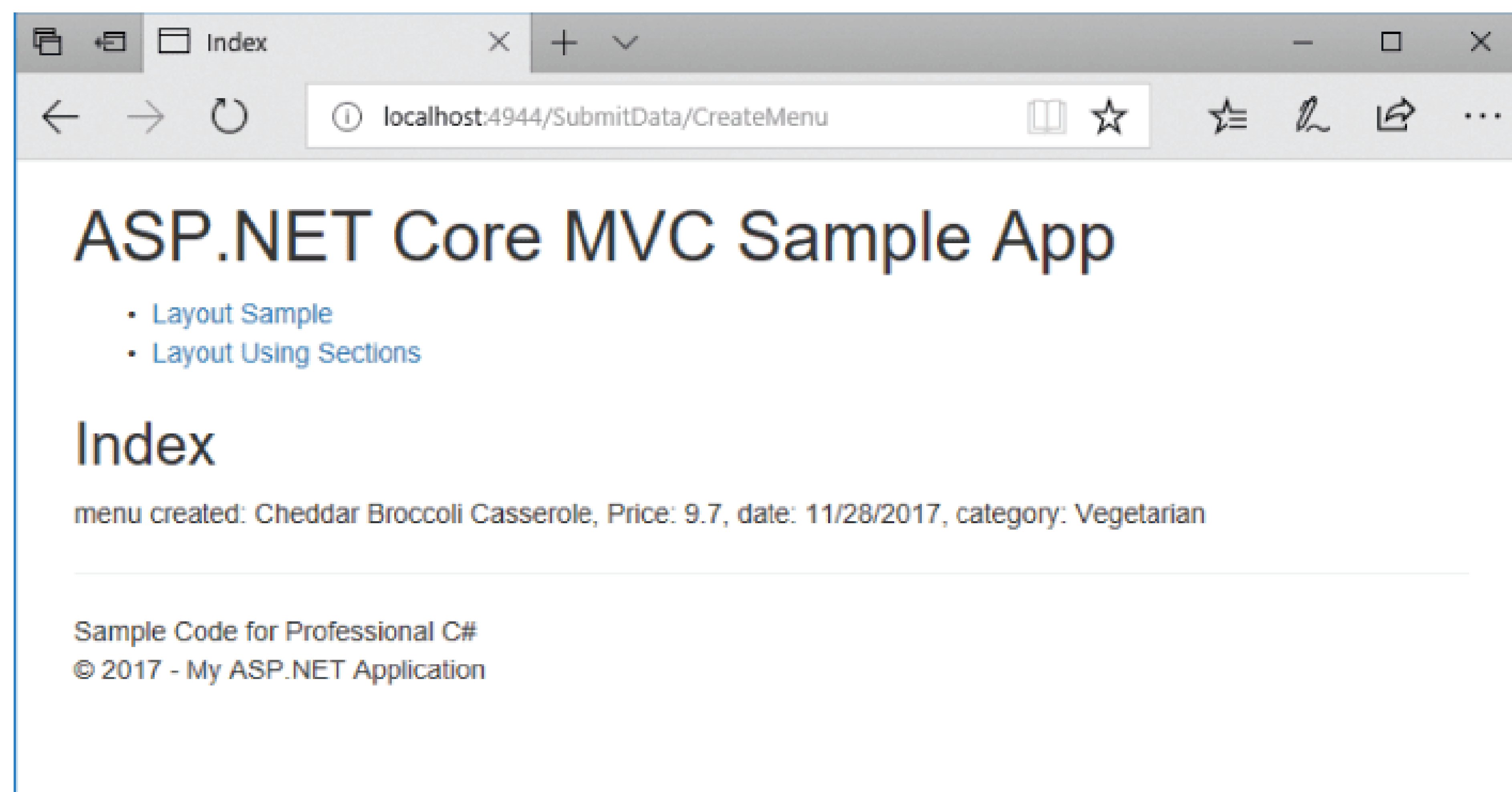



图 31-8

模型绑定器负责传输 HTTP POST 请求中的数据。模型绑定器实现 IModelBinder 接口。默认情况下，使用 FormCollectionModelBinder 类将输入字段绑定到模型。这个绑定器支持基本类型、模型类(如 Menu 类型)以及实现了 ICollection<T>、IList<T>和 IDictionary<TKey, TValue>的集合。

如果并不是所有参数类型的属性都应从模型绑定器中填充，此时可以使用 Bind 特性。通过这个特性，可以指定一个属性名列表，这些属性被应用于绑定。

还可以使用不带参数的动作方法将输入数据传递给模型，如下面的代码段所示。这段代码创建了 Menu 类的一个新实例，并把这个实例传递给 Controller 基类的 TryUpdateModelAsync 方法。如果在更新后，被更新的模型处于无效状态，TryUpdateModelAsync 就返回 false：

```
[HttpPost]
public async Task<IActionResult> CreateMenu3Result()
{
  var m = new Menu();
  bool updated = await TryUpdateModelAsync<Menu>(m);
  if (updated)
  {
    ViewBag.Info = $"menu created: {m.Text}, Price: {m.Price}, " +
      $"date: {m.Date}, category: {m.Category}";
    return View("Index");
  }
  else
  {
    return View("Error");
  }
}
```

### 31.5.2 注解和验证

可以向模型类型添加一些注解，当更新数据时，会将这些注解用于验证。名称空间 System.ComponentModel.DataAnnotations 中包含的特性可用来为客户端数据指定一些信息，或者用来进行验证。

使用其中的一些特性来修改 Menu 类型：

```
public class Menu
{
  public int Id { get; set; }
  [Required, StringLength(50)]
  public string Text { get; set; }
  [Display(Name="Price"), DisplayFormat(DataFormatString="{0:C}")]
  public double Price { get; set; }
  [DataType(DataType.Date)]
  public DateTime Date { get; set; }
  [StringLength(10)]
  public string Category { get; set; }
}
```

可用于验证的特性包括：用于比较不同属性的 CompareAttribute、用于验证有效信用卡号的 CreditCardAttribute、用来验证电子邮件地址的 EmailAddressAttribute、用来比较输入与枚举值的 EnumDataTypeAttribute 以及用来验证电话号码的 PhoneAttribute。

还可以使用其他特性来获得要显示的值或者用在错误消息中的值，如 DataTypeAttribute 和 DisplayFormatAttribute。

为了使用验证特性，可以在动作方法内使用 ModelState.IsValid 来验证模型的状态，如下所示(代码文件 MVCSampleApp/Controllers/SumitDataController.cs)：

```
[HttpPost]
public IActionResult CreateMenu4(Menu menu)
{
  if (ModelState.IsValid)
  {
    ViewBag.Info = $"menu created: {menu.Text}, Price: {menu.Price}, " +
      $"date: {menu.Date}, category: {menu.Category}";
  }
  else
  {
    ViewBag.Info = "not valid";
  }
  return View("Index");
}
```

如果使用工具生成的模型类，则很难给属性添加特性。工具生成的类被定义为部分类，可以通过为其添加属性和方法、实现额外的接口或者实现它们使用的部分方法来扩展这些类。对于已有的属性和方法，如果不能修改类型的源代码，则是不能添加特性的。但是在这种情况下，还是可以利用一些帮助。现在假定 Menu 类是一个工具生成的部分类。可以用一个不同名的新类(如 MenuMetadata)定义与实体类相同的属性并添加注解，如下所示(代码文件 MVCSampleApp/Models/MenuMetadata.cs)：

```
public class MenuMetadata
{
  public int Id { get; set; }
  [Required, StringLength(25)]
  public string Text { get; set; }
  [Display(Name="Price"), DisplayFormat(DataFormatString="{0:C}")]
  public double Price { get; set; }
  [DataType(DataType.Date)]
  public DateTime Date { get; set; }
  [StringLength(10)]
  public string Category { get; set; }
}
```

MenuMetadata 类必须链接到 Menu 类。对于工具生成的部分类，可以在同一个名称空间中创建另一个部分类型，将 ModelMetadataType 特性添加到创建该连接的类型定义中：

```
[ModelMetadataType(typeof(MenuMetadata))]
public partial class Menu
{
}
```

HTML 辅助方法也可以使用注解来向客户端添加信息。

# 31.6　使用 HTML Helper

HTML Helper 是创建 HTML 代码的 Helper。可以在视图中通过 Razor 语法直接使用它们。

Html 是视图基类 RazorPage 的一个属性，它的类型是 IHtmlHelper。HTML Helper 被实现为扩展方法，用于扩展 IHtmlHelper 接口。

类 InputExtensions 定义了用于创建复选框、密码控件、单选按钮和文本框控件的 HTML Helper。Helper Action 和 RenderAction 由类 ChildActionExtensions 定义。用于显示的 Helper 由类 DisplayExtensions 定义。用于 HTML 表单的 Helper 由类 FormExtensions 定义。

接下来就看一些使用 HTML Helper 的例子。

## 31.6.1　简单的 Helper

下面的代码段使用了 HTML Helper BeginForm、Label 和 CheckBox。BeginForm 开始一个表单元素。还有一个用于结束表单元素的 EndForm。示例使用了 BeginForm 方法返回的 MvcForm 所实现的 IDisposable 接口。在释放 MvcForm 时，会调用 EndForm。因此，可以将 BeginForm 方法放在一条 using 语句中，在闭花括号处结束表单。DisplayName 方法直接返回参数的内容，CheckBox 是一个 input 元素，其 type 特性被设置为 checkbox(代码文件 MVCSampleApp/Views/HtmlHelpers/SimpleHelper.cshtml)：

```
@using (Html.BeginForm()) {
  @Html.DisplayName("Check this (or not)")
  @Html.CheckBox("check1")
}
```

得到的 HTML 代码如下所示。CheckBox 方法创建了两个同名的 input 元素，其中一个设置为隐藏。其原因是，如果一个复选框的值为 false，那么浏览器不会把与之对应的信息放到表单内容中传递给服务器。只有选中的复选框的值才会传递给服务器。这种 HTML 特征在自动绑定到动作方法的参数时会产生问题。简单的解决办法是使用 Helper CheckBox。该方法会创建一个同名但被隐藏的 input 元素，并将其设为 false。如果没有选中该复选框，则会把隐藏的 input 元素传递给服务器，绑定 false 值。如果选中了复选框，则同名的两个 input 元素都会传递给服务器。第一个 input 元素设为 true，第二个设为 false。在自动绑定时，只选择第一个 input 元素进行绑定：

```
<form action="/HtmlHelpers/SimpleHelper" method="post">
  Check this (or not)
  <input id="check1" name="check1" type="checkbox" value="true" />
  <input name="check1" type="hidden" value="false" />
</form>
```

## 31.6.2　使用模型数据

Helper 可以使用模型数据。下例创建了一个 Menu 对象。本章前面在 Models 目录中声明了此类型。然后，将该 Menu 对象作为模型传递给视图(代码文件 MVCSampleApp/Controllers/HTML HelpersController.cs)：

```
public IActionResult HelperWithMenu() => View(GetSampleMenu());
private Menu GetSampleMenu() =>
  new Menu
  {
    Id = 1,
    Text = "Schweinsbraten mit Knödel und Sauerkraut",
    Price = 6.9,
    Date = new DateTime(2017, 11, 14),
```

```
    Category = "Main"
  };
```

视图有一个模型定义为 Menu 类型。与前例一样，HTML Helper DisplayName 只是返回参数的文本。Display 方法使用一个表达式作为参数，其中以字符串格式传递一个属性名。该方法试图找出具有这个名称的属性，然后使用属性存取器来返回该属性的值(代码文件 MVCSampleApp/Views /HTMLHelpers/HelperWithMenu.cshtml)：

```
@model MVCSampleApp.Models.Menu
@{
  ViewBag.Title = "HelperWithMenu";
}
<h2>Helper with Menu</h2>
@Html.DisplayName("Text:")
@Html.Display("Text")
<br />
@Html.DisplayName("Category:")
@Html.Display("Category")
```

在得到的 HTML 代码中，可以从调用 DisplayName 和 Display 方法的输出中看到这一点：

```
Text:
Schweinsbraten mit Kn&#246;del und Sauerkraut
<br />
Category:
Main
```

**注意：**
Helper 也提供强类型化方法来访问模型成员，如 31.6.5 节“强类型化的 Helper”所示。

### 31.6.3 定义 HTML 特性

大多数 HTML Helper 都有一些可传递任何 HTML 特性的重载版本。例如，下面的 TextBox 方法创建一个文本类型的 input 元素。其第一个参数定义了文本框的名称，第二个参数定义了文本框设置的值。TextBox 方法的第三个参数是 object 类型，允许传递一个匿名类型，在其中将每个属性改为 HTML 元素的一个特性。在这里，input 元素的结果是将 required 特性设为 required，将 maxlength 特性设为 15，将 class 特性设为 CSSDemo。因为 class 是 C#的一个关键字，所以不能直接设为一个属性，而是要加上@作为前缀，以生成用于 CSS 样式的 class 特性：

```
@Html.TextBox("text1", "input text here",
  new { required="required", maxlength=15, @class="CSSDemo" });
```

得到的 HTML 输出如下所示：

```
<input name="text1" class="CSSDemo" id="text1" maxlength="15"
  required="required" type="text" value="input text here" />
```

### 31.6.4 创建列表

为显示列表，需要使用 DropDownList 和 ListBox 等 Helper。这些 Helper 会创建 HTML select 元素。

在控制器内，首先创建一个包含键和值的字典。然后使用自定义扩展方法 ToSelectListItems，将该字典转换为 SelectListItem 的列表。DropDownList 和 ListBox 方法使用了 SelectListItem 集合(代码文件 MVCSampleApp/Controllers/HTMLHelpersController.cs)：

```
public IActionResult HelperList()
{
  var cars = new Dictionary<int, string>();
  cars.Add(1, "Red Bull Racing");
  cars.Add(2, "McLaren");
  cars.Add(3, "Mercedes");
  cars.Add(4, "Ferrari");
  return View(cars.ToSelectListItems(4));
}
```

自定义扩展方法 ToSelectListItems 在扩展了 IDictionary<int, string>的 SelectListItemsExtensions 类中定义，IDictionary<int, string>是 cars 集合中的类型。在其实现中，只是为字典中的每一项返回一个新的 SelectListItem 对象(代码文件 MVCSampleApp/Extensions/SelectListItemsExtensions.cs)：

```
public static class SelectListItemsExtensions
{
  public static IEnumerable<SelectListItem> ToSelectListItems(
    this IDictionary<int, string> dict, int selectedId) =>
      dict.Select(item =>
        new SelectListItem
        {
          Selected = item.Key == selectedId,
          Text = item.Value,
          Value = item.Key.ToString()
        });
}
```

在视图中，Helper DropDownList 直接访问从控制器返回的模型(代码文件 MVCSampleApp/Views/HTMLHelpers/HelperList.cshtml)：

```
@{
  ViewBag.Title = "Helper List";
}
@model IEnumerable<SelectListItem>
<h2>Helper2</h2>
@Html.DropDownList("carslist", Model)
```

得到的 HTML 创建了一个 select 元素，该元素包含通过 SelectListItem 创建的一些 option 子元素。这些 HTML 还定义了从控制器中返回的选中项：

```
<select id="carslist" name="carslist">
  <option value="1">Red Bull Racing</option>
  <option value="2">McLaren</option>
  <option value="3">Mercedes</option>
  <option selected="selected" value="4">Ferrari</option>
</select>
```

### 31.6.5 强类型化的 Helper

HTML Helper 提供了强类型化的方法来访问从控制器传递的模型。这些方法都带有后缀 For。例如，可以使用 TextBoxFor 代替 TextBox 方法。

下面的示例再次使用返回单个实体的控制器(代码文件 MVCSampleApp/Controllers/HTMLHelpersController.cs)：

```
public IActionResult StronglyTypedMenu() => View(GetSampleMenu());
```

视图使用 Menu 类型作为模型，所以可以使用 DisplayNameFor 和 DisplayFor 方法直接访问 Menu 属性。DisplayNameFor 默认返回属性名(在这里是 Text 属性)，DisplayFor 返回属性值(代码文件 MVCSampleApp/Views/HTMLHelpers/StronglyTypedMenu.cshtml)：

```
@using MVCSampleApp.Models
@model Menu
@Html.DisplayNameFor(m => m.Text)
<br />
@Html.DisplayFor(m => m.Text)
```

类似地，可以使用 Html.TextBoxFor(m => m.Text)，它返回一个允许设置模型的 Text 属性的 input 元素。该方法还使用了添加到 Menu 类型的 Text 属性的注解。Text 属性添加了 Required 和 MaxStringLength 特性，所以 TextBoxFor 方法会返回 data-val-length、data-val-length-max 和 data-val-required 特性：

```
<input data-val="true"
  data-val-length="The field Text must be a string with a maximum length of 50."
  data-val-length-max="50"
  data-val-required="The Text field is required."
  id="FileName_Text" name="Text"
  type="text"
  value="Schweinsbraten mit Knödel und Sauerkraut" />
```

### 31.6.6 编辑器扩展

除了为每个属性使用至少一个 Helper 外，EditorExtensions 类中的 Helper 还给一个类型的所有属性提供了一个编辑器。

使用与前面相同的 Menu 模型，通过方法 Html.EditorFor(m => m)构建一个用于编辑菜单的完整 UI。该方法调用的结果如图 31-9 所示。

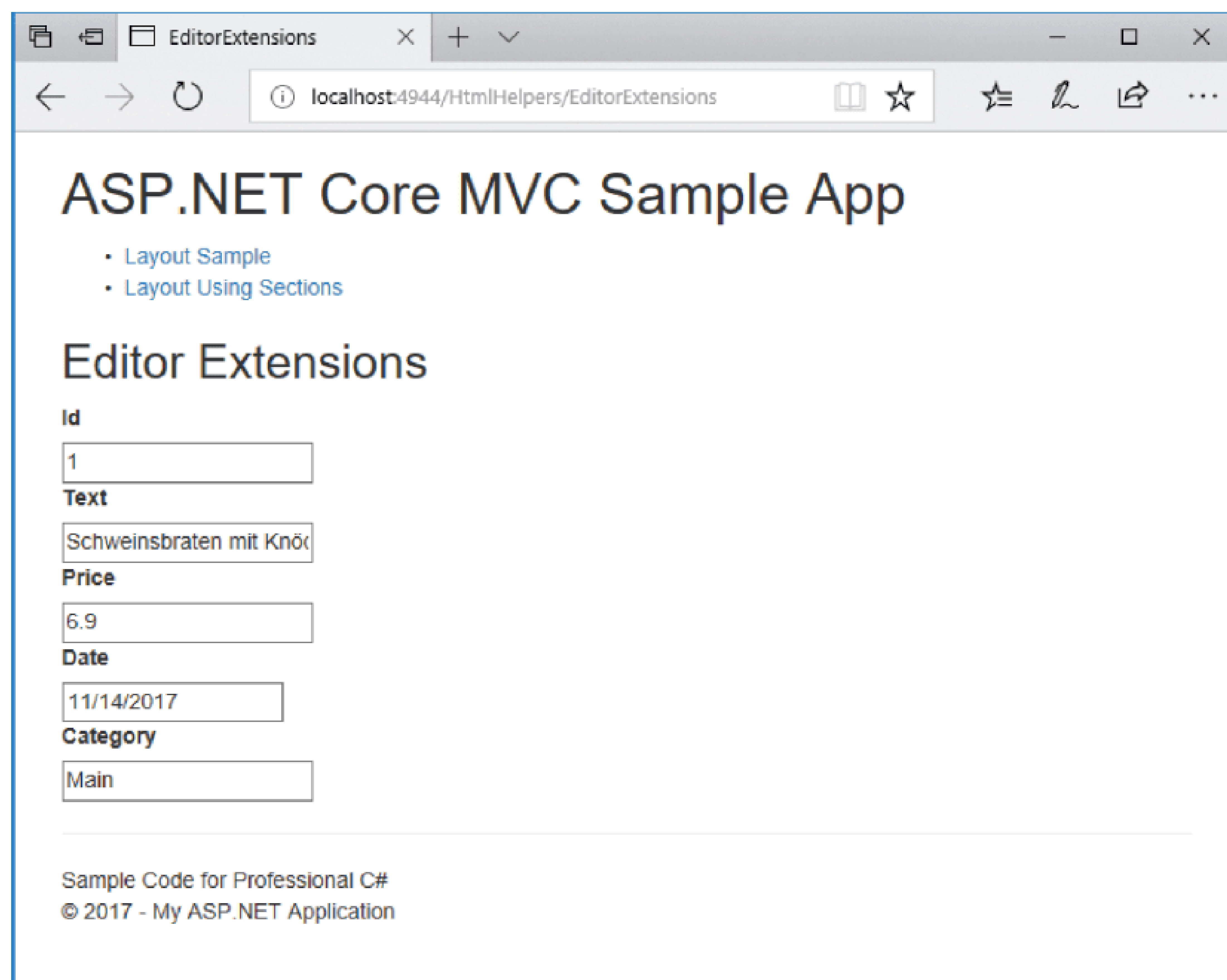



图 31-9

除了使用 Html.EditorFor(m => m)，还可以使用 Html.EditorForModel。EditorForModel 方法会使用视图的模型，不需要显式指定模型。EditorFor 在使用其他数据源(例如模型提供的属性)方面更加灵活，EditorForModel 需要添加的参数更少。

### 31.6.7 实现模板

使用模板是扩展 HTML Helper 的结果的一种好方法。模板是 HTML 辅助方法被隐式或显式使用的一个简单视图，它们存储在特殊的文件夹中。显示模板存储在视图文件夹下的 DisplayTemplates 文件夹中(如 Views/HtmlHelpers/DisplayTemplates)，或者存储在共享文件夹中(如 Shared/DisplayTemplates)。共享文件夹由全部视图使用，特定的视图文件夹则只有该文件夹中的视图可以使用。编辑器模板存储在 EditorTemplates 文件夹中。

现在看一个示例。在 Menu 类型中，Date 属性有一个注解 DataType，其值为 DataType.Date。指定该特性时，DateTime 类型默认并不会显示为日期加时间的形式，而是显示为短日期格式(代码文件 MVCSampleApp/Models/Menu.cs)：

```
public class Menu
{
  public int Id { get; set; }

  [Required, StringLength(50)]
  public string Text { get; set; }
```

```
    [Display(Name="Price"), DisplayFormat(DataFormatString="{0:c}")]
    public double Price { get; set; }

    [DataType(DataType.Date)]
    public DateTime Date { get; set; }

    [StringLength(10)]
    public string Category { get; set; }
}
```

现在为日期创建了模板。这里使用了长日期字符串格式 D 来返回 Model，将这个日期字符串格式 D 嵌入在 CSS 类为 markRed 的 div 标记内(代码文件 MVCSampleApp/Views/HTMLHelpers/DisplayTemplates/Date.cshtml)：

```
<div class="markRed">
  @string.Format("{0:D}", Model)
</div>
```

CSS 类 markRed 在样式表中定义，用于设置红色(代码文件 MVCSampleApp/wwwroot/css/Site.css)：

```
.markRed {
  color: #f00;
}
```

现在像 DisplayForModel 这样用于显示的 HTML Helper 可以使用已定义的模板。模型的类型是 Menu，所以 DisplayForModel 方法会显示 Menu 类型的所有属性。对于 Date，它找到模板 Date.cshtml，所以会使用该模板以 CSS 样式显示长日期格式的日期(代码文件 MVCSampleApp/Views/HTML Helpers/Display.cshtml)：

```
@model MVCSampleApp.Models.Menu
@{
  ViewBag.Title = "Display";
}
<h2>@ViewBag.Title</h2>
@Html.DisplayForModel()
```

如果在同一个视图内，某个类型应该有不同的表示，则可以为模板文件使用其他名称。之后就可以使用 UIHint 特性来指定这个模板的名称，或者使用辅助方法的模板参数指定模板。

## 31.7　Tag Helper

ASP.NET Core MVC 提供了一种新技术，可以用来代替 HTML Helper：Tag Helper。对于 Tag Helper，不要编写混合了 HTML 的 C#代码，而是使用在服务器上解析的 HTML 特性和元素。如今许多 JavaScript 库用自己的特性(如 Angular)扩展了 HTML，所以可以很方便地把自定义的 HTML 特性用于服务器端技术。许多 ASP.NET Core MVC Tag Helper 都有前缀 asp-，所以很容易看出在服务器上解析了什么。这些特性不发送给客户端，而是在服务器上解析，生成 HTML 代码。

### 31.7.1　激活 Tag Helper

要使用 ASP.NET Core MVC Tag Helper，需要调用 addTagHelper 来激活标记。它的第一个参数定义了要使用的类型(*会打开程序集的所有 Tag Helper)；第二个参数定义了 Tag Helper 的程序集。使用 removeTagHelper，会再次取消激活 Tag Helper。取消激活 Tag Helper 可能很重要，例如不与脚本库发生命名冲突。给内置的 Tag Helper 使用 asp-前缀，发生冲突的可能性最小，但如果内置的 Tag Helper 与其他的 Tag Helper 同名，其他的 Tag Helper 有用于脚本库的 HTML 特性，就很容易发生冲突。

为了使 Tag Helper 可用于所有的视图，应把 addTagHelper 语句添加到共享文件_ViewImports.cshtml 中(代码文件 MVCSampleApp/Views/_ViewImports.cshtml)：

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

### 31.7.2　使用锚定 Tag Helper

下面从扩展锚元素 a 的 Tag Helper 开始。Tag Helper 的示例控制器是 TagHelpersController。Index 动作方法

返回一个视图，用来显示锚 Tag Helper (代码文件 MVCSampleApp/Controllers/TagHelpersController.cs)：

```
public class TagHelpersController : Controller
{
  public IActionResult Index() => View();
  //...
}
```

锚 Tag Helper 定义了 asp-controller 和 asp-action 特性。之后，控制器和动作方法用来建立锚元素的 URL。在第二个和第三个例子中，不需要控制器，因为视图来自相同的控制器(代码文件 MVCSampleApp/Views/TagHelpers/Index.cshtml )：

```
<a asp-controller="Home" asp-action="Index">Home</a>
<br />
<a asp-action="LabelHelper">Label Tag Helper</a>
<br />
<a asp-action="InputTypeHelper">Input Type Tag Helper</a>
```

以下代码段显示了生成的 HTML 代码。asp-controller 和 asp-action 特性为 a 元素生成了 href 特性。在第一个示例中，为了访问 Home 控制器中的 Index 动作方法，因为它们都是路由定义的默认值，所以结果中只需要指向“/”的 href。指定 asp-action LabelHelper 时，href 指向/TagHelpers/LabelHelper，即当前控制器中的动作方法 LabelHelper：

```
<a href="/">Home</a>
<br />
<a href="/TagHelpers/LabelHelper">Label Tag Helper</a>
<br />
<a href="/TagHelpers/InputTypeHelper">Input Type Tag Helper</a>
```

### 31.7.3 使用 Label Tag Helper

下面的代码段展示了 Label Tag Helper 的功能，其中动作方法 LabelHelper 把 Menu 对象传递到视图(代码文件 MVCSampleApp/Controllers/TagHelpersController.cs)：

```
public IActionResult LabelHelper() => View(GetSampleMenu());
private Menu GetSampleMenu() =>
  new Menu
  {
    Id = 1,
    Text = "Schweinsbraten mit Knödel und Sauerkraut",
    Price = 6.9,
    Date = new DateTime(2018, 10, 5),
    Category = "Main"
  };
}
```

Menu 类应用了一些数据注解，用来影响 Tag Helper 的结果。看一看 Text 属性的 Display 特性。它将 Display 特性的 Name 属性设置为“Menu”(代码文件 MVCSampleApp/Models/Menu.cs)：

```
public class Menu
{
  public int Id { get; set; }

  [Required, StringLength(50)]
  [Display(Name = "Menu")]
  public string Text { get; set; }

  [Display(Name = "Price"), DisplayFormat(DataFormatString = "{0:C}")]
  public double Price { get; set; }

  [DataType(DataType.Date)]
  public DateTime Date { get; set; }

  [StringLength(10)]
  public string Category { get; set; }
}
```

视图利用了应用于标签控件的asp-for特性。用于此特性的值是视图模型的一个属性。在Visual Studio 2017中，可以使用智能感知来访问Text、Price和Date属性(代码文件MVCSampleApp/Views/TagHelpers/LabelHelper.cshtml)：

```
@model MVCSampleApp.Models.Menu
@{
  ViewBag.Title = "Label Tag Helper";
}
<h2>@ViewBag.Title</h2>
<label asp-for="Text"></label>
<br/>
<label asp-for="Price"></label>
<br />
<label asp-for="Date"></label>
```

在生成的 HTML 代码中，可以看到 for 特性，它引用的元素与属性同名，内容是属性名或 Display 特性的值。还可以使用此特性本地化值：

```
<label for="Text">Menu</label>
<br/>
<label for="Price">Price</label>
<br />
<label for="Date">Date</label>
```

### 31.7.4　使用 Input Tag Helper

HTML 标签通常与 input 元素相关。下面的代码段说明了使用 input 元素和 Tag Helper 会生成什么(代码文件MVCSampleApp/Views/TagHelpers/InputHelper.cshtml)：

```
<div>
  <label asp-for="Text"></label>
  <input asp-for="Text"/>
</div>
<div>
  <label asp-for="Price"></label>
  <input asp-for="Price" />
</div>
<div>
  <label asp-for="Date"></label>
  <input asp-for="Date" />
</div>
```

检查生成的 HTML 代码的结果，会发现 input 类型的 Tag Helper 根据属性的类型创建一个 type 特性，它们也应用了 DateType 特性。属性 Price 的类型是 double，得到一个数字输入类型。因为 Date 属性的 DataType 应用了 DataType.Date 值，所以输入类型是日期。此外，还创建了 data-val-length、data-val-length-max 和 data-val-required 特性，用于注解：

```
<div>
  <label for="Text">Text</label>
  <input type="text" data-val="true"
    data-val-length="The field Text must be a string with a maximum length of 50."
    data-val-length-max="50" data-val-required="The Text field is required."
    id="Text" name="Text"
    value="Schweinsbraten mit Kn&#xF6;del und Sauerkraut" />
</div>
<div>
  <label for="Price">Price</label>
  <input type="text" data-val="true"
    data-val-number="The field Price must be a number."
    data-val-required="The Price field is required." id="Price"
    name="Price" value="6.9" />
</div>
<div>
  <label for="Date">Date</label>
  <input type="date" data-val="true"
    data-val-required="The Date field is required." id="Date" name="Date"
    value="2018-10-05" />
</div>
```

现代浏览器给 HTML 5 输入控件(如日期控件)提供了特别的外观。Microsoft Edge 的输入日期控件如图 31-10

所示，它不同于其他浏览器。

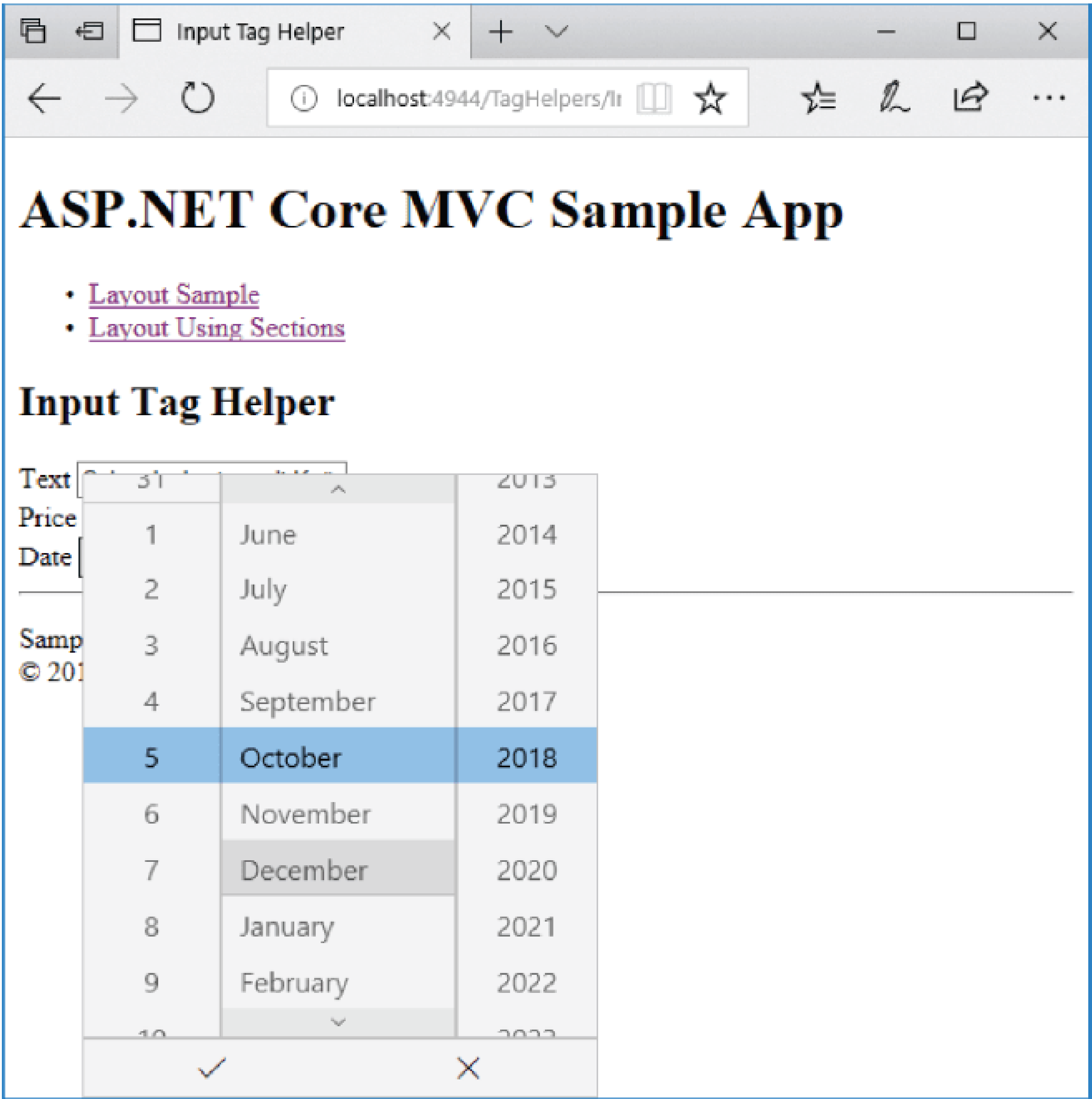



图 31-10

## 31.7.5 使用表单进行验证

为了把数据发送到服务器，输入字段需要用表单包围起来。表单的 Tag Helper 使用 asp-method 和 asp-controller 定义了 action 特性。对于 input 控件，验证信息是由这些控件定义的。需要显示验证错误。为了显示，验证消息 Tag Helper 用 asp-validation-for 扩展了 span 元素(代码文件 MVCSampleApp/Views/TagHelpers/FormHelper.cshtml)：

```
<form method="post" asp-method="FormHelper">
  <input asp-for="Id" hidden="hidden" />
  <hr />
  <label asp-for="Text"></label>
  <div>
    <input asp-for="Text" />
    <span asp-validation-for="Text"></span>
  </div>
  <br />
    <label asp-for="Price"></label>
    <div>
    <input asp-for="Price" />
   <span asp-validation-for="Price"></span>
</div>
<br />
<label asp-for="Date"></label>
<div>
<input asp-for="Date" />
<span asp-validation-for="Date"></span>
</div>
<label asp-for="Category"></label>
<div>
<input asp-for="Category" />
<span asp-validation-for="Category"></span>
</div>
<input type="submit" value="Submit" />
</form>
```

控制器检查 ModelState，验证接收数据是否正确。如果不正确，就再次显示同样的视图(代码文件 MVCSampleApp/Controllers/TagHelpersController.cs)：

```
public IActionResult FormHelper() => View(GetSampleMenu());

[HttpPost]
public IActionResult FormHelper(Menu m)
{
  if (!ModelState.IsValid)
  {
    return View(m);
  }
  return View("ValidationHelperResult", m);
}
```

运行应用程序时，错误信息如图 31-11 所示。

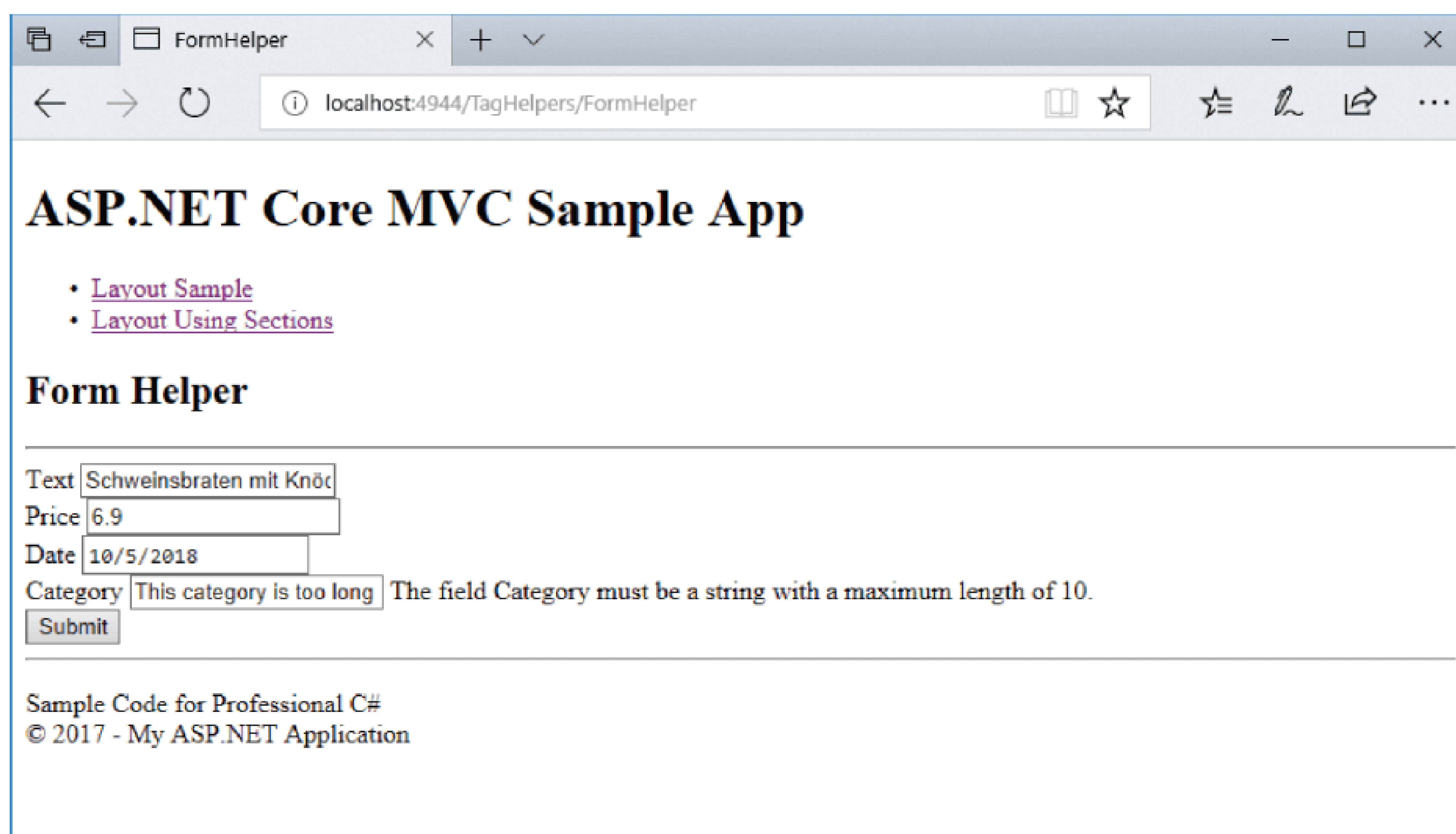



图 31-11

## 31.7.6　environment Tag Helper

environment Tag Helper 允许用户界面的特定部分仅在特定的环境中可用。第 30 章展示了如何使用具有 EnvironmentalName 属性、扩展方法 IcDevelopment、IsStaging 和 IsProduction 的 IHostingEnvironment 接口区分不同的环境。environment Tag Helper 支持类似的功能。

在下面的代码片段中，只有在使用 include 特性在 Development 和 Staging 环境中运行时，才使用第一个环境标记来显示包含的内容。第二个环境标记不包括 Production 环境；因此，包含的内容在 Development、Staging 和其他可以配置的环境中显示(代码文件 MVCSampleApp/Views/TagHelpers/EnvironmentHelper.cshtml)：

```
<environment include="Development, Staging">
  <div>This shows up for the <strong>Development</strong>
    and <strong>Staging</strong> environments</div>
</environment>
<environment exclude="Production">
  <div>Not visible in the <strong>Production</strong> environment</div>
</environment>
```

可以通过设置环境变量 ASPNETCORE_ENVIRONMENT 来配置活动环境，这可以通过 Visual Studio 中的项目设置完成(参见图 31-12)，也可以通过配置 launchsettings.json 文件来完成。

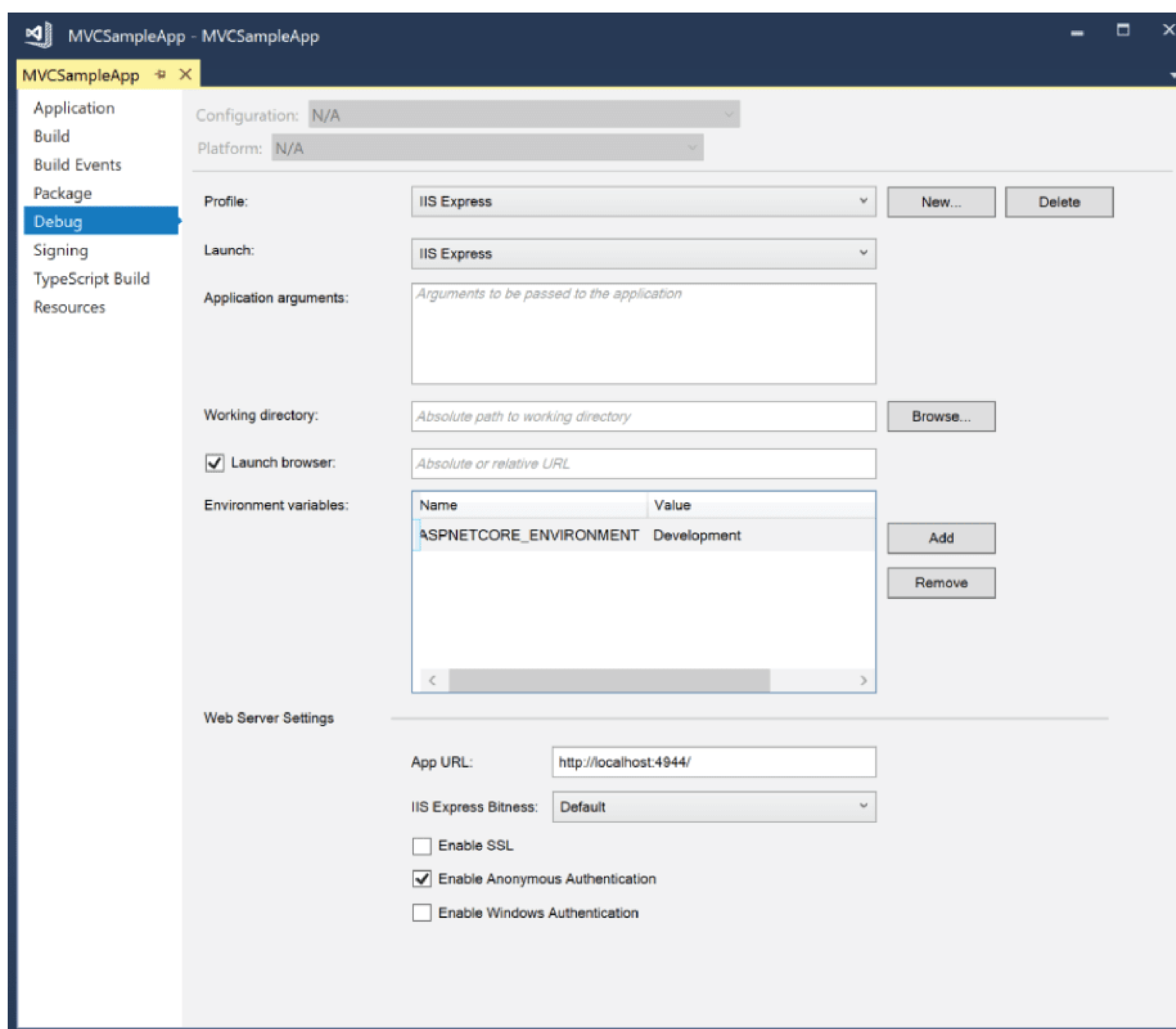



图 31-12

## 31.7.7 创建自定义 Tag Helper

除了使用预定义的 Tag Helper 之外，也可以创建自定义的 Tag Helper。第一个自定义 Tag Helper 在 NuGet 包 Markdig 的帮助下将 Markdown 代码转换为 HTML。

> **注意：**
> Markdown 是一种可以通过文本编辑器轻松创建的标记语言。Markdown 很容易转换成 HTML。请登录 https://csharp.christiannagel.com/2016/07/03/markdown/阅读博客文章“Using Markdown”，了解使用.NET 和 Markdown 的信息。

Tag Helper MarkdownTagHelper 在一个名为 TagHelperSamples 的.NET Core 库中实现，该库引用了 NuGet 包 Microsoft.AspNetCore.`All` 和 Markdig。

下面的代码片段显示了 MarkdownTagHelper 的类声明。Tag Helper 派生自基类 TagHelper。特性 HtmlTargetElement 定义了用于指定 Tag Helper 的元素或特性名称。这个 Tag Helper 可以与 markdown 元素一起使用，也可以与在 div 元素中使用的 markdownfile 特性一起使用。如果元素需要自闭(枚举值 WithoutEndTag，或者使用 NormalOrSelfClosing 允许结束标记或自闭)，那么 TagStructure 特性允许进行配置(代码文件 TagHelperSamples/MarkdownTagHelper.cs.cs)：

```
[HtmlTargetElement("markdown",
  TagStructure = TagStructure.NormalOrSelfClosing)]
[HtmlTargetElement(Attributes = "markdownfile")]
public class MarkdownTagHelper : TagHelper
{
```

```
  //...
}
```

Tag Helper 可以使用依赖注入。因为 MarkdownTagHelper 需要 wwwroot 文件的目录，这个目录是从 IHostingEnvironment 接口中返回的，所以这个接口被注入到构造函数中：

```
private readonly IHostingEnvironment _env;
public MarkdownTagHelper(IHostingEnvironment env) => _env = env;
```

Tag Helper 的属性在用 HtmlAttributeName 特性注释时自动应用于基础结构。在这里，属性 MarkdownFile 从 markdownfile 特性中获取其值：

```
[HtmlAttributeName("markdownfile")]
public string MarkdownFile { get; set; }
```

接下来了解这个 Tag Helper 的主要功能。Tag Helper 需要重写其中一个方法 Process 或 ProcessAsync。当需要异步功能时，将使用 ProcessAsync 方法，而如果只调用同步方法，则可以使用 Process 方法。下面的代码片段重写了 ProcessAsync 方法，因为在实现中使用了异步方法 GetChildContentAsync。通过实现，可以考虑 MarkdownTagHelper 的两个不同用法。一种用法是指定 markdown 元素，其内容作为元素的子元素，另一种用法是引用 Markdown 文件的 Markdownfile 特性。

如果使用了特性 markdownfile，就设置 MarkdownFile 属性，从而读取此属性指定的文件，并将内容写入 markdown 变量。文件的目录通过类型为 IHostingEnvironment 的_env 变量检索。这个接口定义了 WebRootPath 属性，该属性返回 Web 文件的根路径。

如果没有设置 MarkdownFile 属性，而是使用 markdown 元素，则读取该元素的内容。使用 TagHelperOutput 可以访问 markdown 中指定的元素内容。要检索内容，需要调用 GetChildContentAsync 方法，在这个方法返回后，需要调用 GetContent 方法，最终返回 HTML 页面中指定的内容。使用 Markdig 库的 Markdown 类，将 Markdown 内容转换为 HTML。然后将调用 SetHtmlContent 方法，此 HTML 代码放入 TagHelperOutput 的内容中(代码文件 TagHelperSamples/MarkdownTagHelper.cs)：

```
public override async Task ProcessAsync(TagHelperContext context,
  TagHelperOutput output)
{
  if (context == null) throw new ArgumentNullException(nameof(context));
  if (output == null) throw new ArgumentNullException(nameof(output));

  string markdown = string.Empty;
  if (MarkdownFile != null)
  {
    string filename = Path.Combine(_env.WebRootPath, MarkdownFile);
    markdown = File.ReadAllText(filename);
  }
  else
  {
    markdown = (await output.GetChildContentAsync()).GetContent();
  }
  output.Content.SetHtmlContent(Markdown.ToHtml(markdown));
}
```

在创建 MarkdownTagHelper 之后，可以在 CSHTML 文件中使用它。首先，@addTagHelper 添加来自库 TagHelperSamples 的所有 Tag Helper。在 HTML 代码中，使用 markdown 元素。这个元素包含一小段 Markdown 语法，包括标题 2、一个链接和一个列表(代码文件 MVCSampleApp/Views/TagHelpers/Markdown.cshtml)：

```
@addTagHelper *, TagHelperSamples

<h2>Markdown Sample</h2>

<markdown>
## This is simple Markdown

[C# Blog](https://csharp.christiannagel.com)

* one
* two
* three
</markdown>
```

运行应用程序时，markdown 语法被转换为 HTML，如图 31-13 所示。

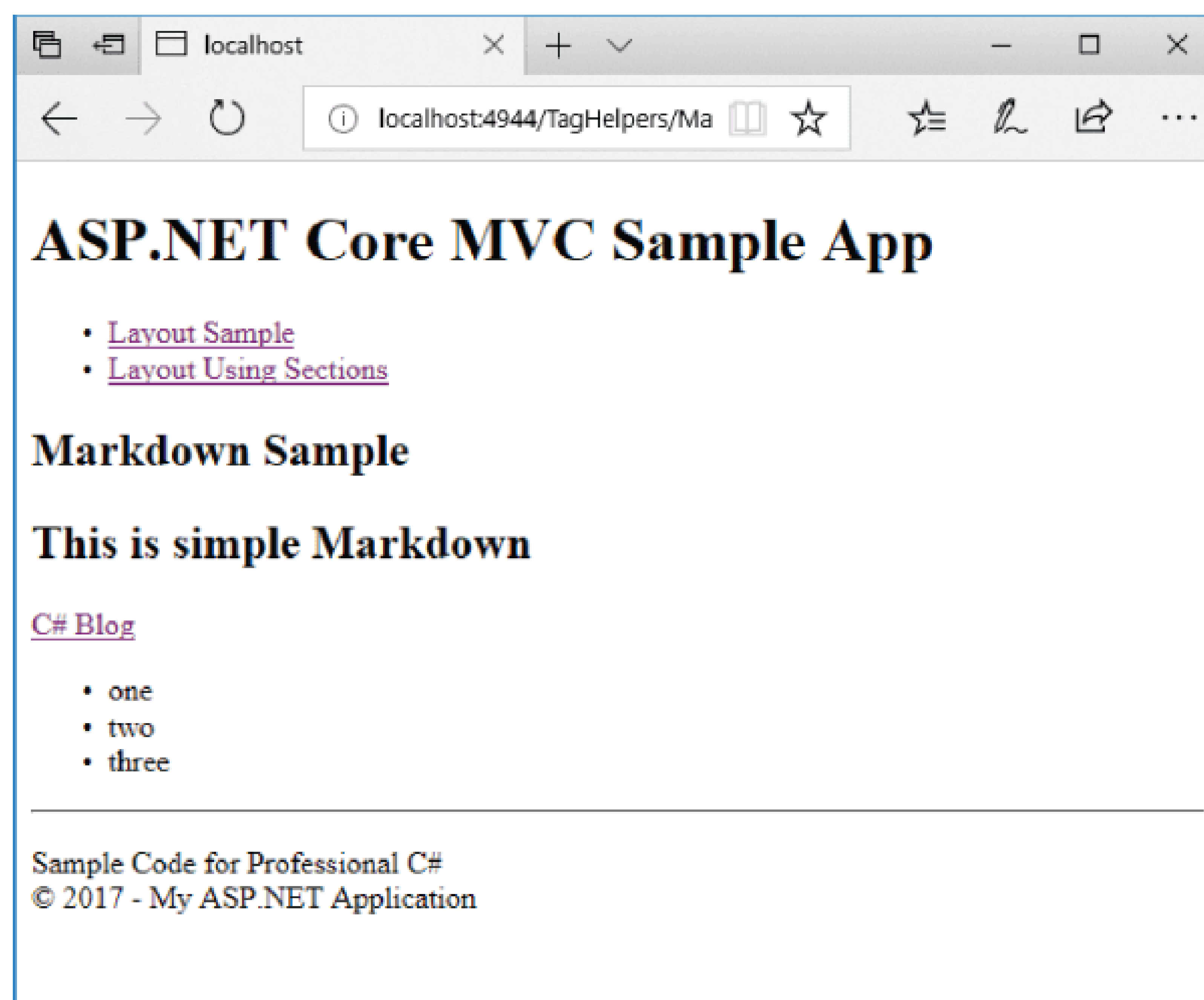



图 31-13

现在，通过创建文件 Sample.md(包含与前面所示相同的 Markdown 内容)，并引用 markdownfile 特性中的文件可以实现相同的功能(代码文件 MVCSampleApp/Views/TagHelpers/MarkdownAttribute.cshtml)：

```
<div markdownfile="Sample.md"></div>
```

这样，MarkdownTagHelper 的属性 MarkdownFile 就设置好了，因此会读取 markdown 文件。

### 31.7.8 用 Tag Helper 创建元素

本节建立的示例自定义 Tag Helper 扩展了 HTML 元素 table，为列表中的每项显示一行，为每个属性显示一列。把数据信息的模型传递给 Tag Helper，Tag Helper 就会动态创建 table、tr、th 和 td 元素。应创建的信息使用反射来完成。类似这样的功能也可以在视图组件中实现，视图 Helper 可以与 Tag Helper 一起使用。本节详细介绍如何创建更复杂的 Tag Helper，使用 TagBuilder 类动态创建 HTML 元素。

> **注意：**
> 反射参见第 16 章。

对于本例，控制器实现了方法 CustomHelper，以返回包含 Menu 对象列表的视图(代码文件 MVCSampleApp/Controllers/TagHelpersController.cs)：

```
public IActionResult CustomTable() => View(GetSampleMenus());

private IList<Menu> GetSampleMenus() =>
  new List<Menu>()
  {
    new Menu
    {
      Id = 1,
      Text = "Schweinsbraten mit Knödel und Sauerkraut",
      Price = 8.5,
      Date = new DateTime(2018, 10, 5),
      Category = "Main"
    },
    new Menu
    {
      Id = 2,
      Text = "Erdäpfelgulasch mit Tofu und Gebäck",
      Price = 8.5,
      Date = new DateTime(2018, 10, 6),
```

```
      Category = "Vegetarian"
    },
    new Menu
    {
      Id = 3,
      Text = "Tiroler Bauerngröst'l mit Spiegelei und Krautsalat",
      Price = 8.5,
      Date = new DateTime(2018, 10, 7),
      Category = "Vegetarian"
    }
};
```

Tag Helper 类 TableTagHelper 用 HTML table 元素激活。与前面使用 markup 元素的辅助程序相反，这个辅助程序与有效的 HTML 元素一起使用。HtmlTargetElement 指定 table 和 items 特性来应用这个辅助程序，这个 items 特性用于设置 Items 属性，与 HtmlAttributeName 特性指定 Items 属性一样(代码文件 TagHelperSamples/TableTagHelper.cs)：

```
[HtmlTargetElement("table", Attributes = ItemsAttributeName)]
public class TableTagHelper : TagHelper
{
  private const string ItemsAttributeName = "items";

  [HtmlAttributeName(ItemsAttributeName)]
  public IEnumerable<object> Items { get; set; }
  //...
}
```

Tag Helper 的核心是方法 Process。这次可以使用这个方法的同步变体，因为实现代码中没有使用异步方法。通过方法 Process 的参数，接收一个 TagHelperContext。这个上下文包含应用了 Tag Helper 的 HTML 元素和所有子元素的特性。对于使用 Tag Helper 时指定的表元素，行和列可能已经定义，可以合并该结果与现有的内容。在示例中，这被忽略了，只是把特性放在结果中。结果需要写入第二个参数：TagHelperOutput 对象。为了创建 HTML 代码，使用 TagBuilder 类型。TagBuilder 帮助通过特性创建 HTML 元素，它还处理元素的关闭。为了给 TagBuilder 添加特性，使用 MergeAttributes 方法。这个方法需要一个包含所有特性名称和值的字典。这个字典使用 LINQ 扩展方法 ToDictionary 创建。在 Where 方法中，提取表元素所有已有的特性，但 items 特性除外。items 特性用于通过 Tag Helper 定义项，但以后在客户端不需要它：

```
public override void Process(TagHelperContext context, TagHelperOutput output)
{
  if (context == null) throw new ArgumentNullException(nameof(context));
  if (output == null) throw new ArgumentNullException(nameof(output));

  var table = new TagBuilder("table");
  table.GenerateId(context.UniqueId, "id");
  var attributes = context.AllAttributes
    .Where(a => a.Name != ItemsAttributeName)
    .ToDictionary(a => a.Name);
  table.MergeAttributes(attributes);

  PropertyInfo[] properties = CreateHeading(table);
  //...
}
```

**注意：**
LINQ 参见第 12 章。

接下来，使用 CreateHeading 方法创建表中的第一行。这一行包含一个 tr 元素，作为 table 元素的子元素，它还为每个属性包含 td 元素。为了获得所有的属性名，调用 First 方法，检索集合的第一个对象。使用反射访问该实例的属性，调用 Type 对象上的 GetProperties 方法，把属性的名称写入 HTML 元素 th 的内部文本：

```
private PropertyInfo[] CreateHeading(TagBuilder table)
{
  var tr = new TagBuilder("tr");
  var heading = Items.First();
  PropertyInfo[] properties = heading.GetType().GetProperties();
  foreach (var prop in properties)
  {
    var th = new TagBuilder("th");
```

```
    th.InnerHtml.Append(prop.Name);
    tr.InnerHtml.AppendHtml(th);
  }
  table.InnerHtml.AppendHtml(tr);
  return properties;
}
```

Process 方法的最后一部分遍历集合的所有项，为每一项创建更多的行(tr)。对于每个属性，添加 td 元素，将属性的值写入为内部文本。最后，将所建 table 元素的内部 HTML 代码写到输出：

```
foreach (var item in Items)
{
  tr = new TagBuilder("tr");
  foreach (var prop in properties)
  {
    var td = new TagBuilder("td");
    td.InnerHtml.Append(prop.GetValue(item).ToString());
    tr.InnerHtml.AppendHtml(td);
  }
  table.InnerHtml.AppendHtml(tr);
  }
  output.Content.Append(table.InnerHtml);
}
```

在创建 Tag Helper 之后，创建视图就变得非常简单。定义了模型后，传递程序集的名称，通过 addTagHelper 引用 Tag Helper。使用特性 items 定义一个 HTML 表时，实例化 Tag Helper 本身（代码文件 MVCSampleApp/Views/TagHelpers/CustomHelper.cshtml ）：

```
@model IEnumerable<Menu>
@addTagHelper *, MVCSampleApp
<table items="Model" class="sample"></table>
```

运行应用程序时，表应该如图 31-14 所示。创建了 Tag Helper 后，使用起来很简单。使用 CSS 定义的所有格式仍适用，因为定义的 HTML 表的所有特性仍在生成的 HTML 输出中。

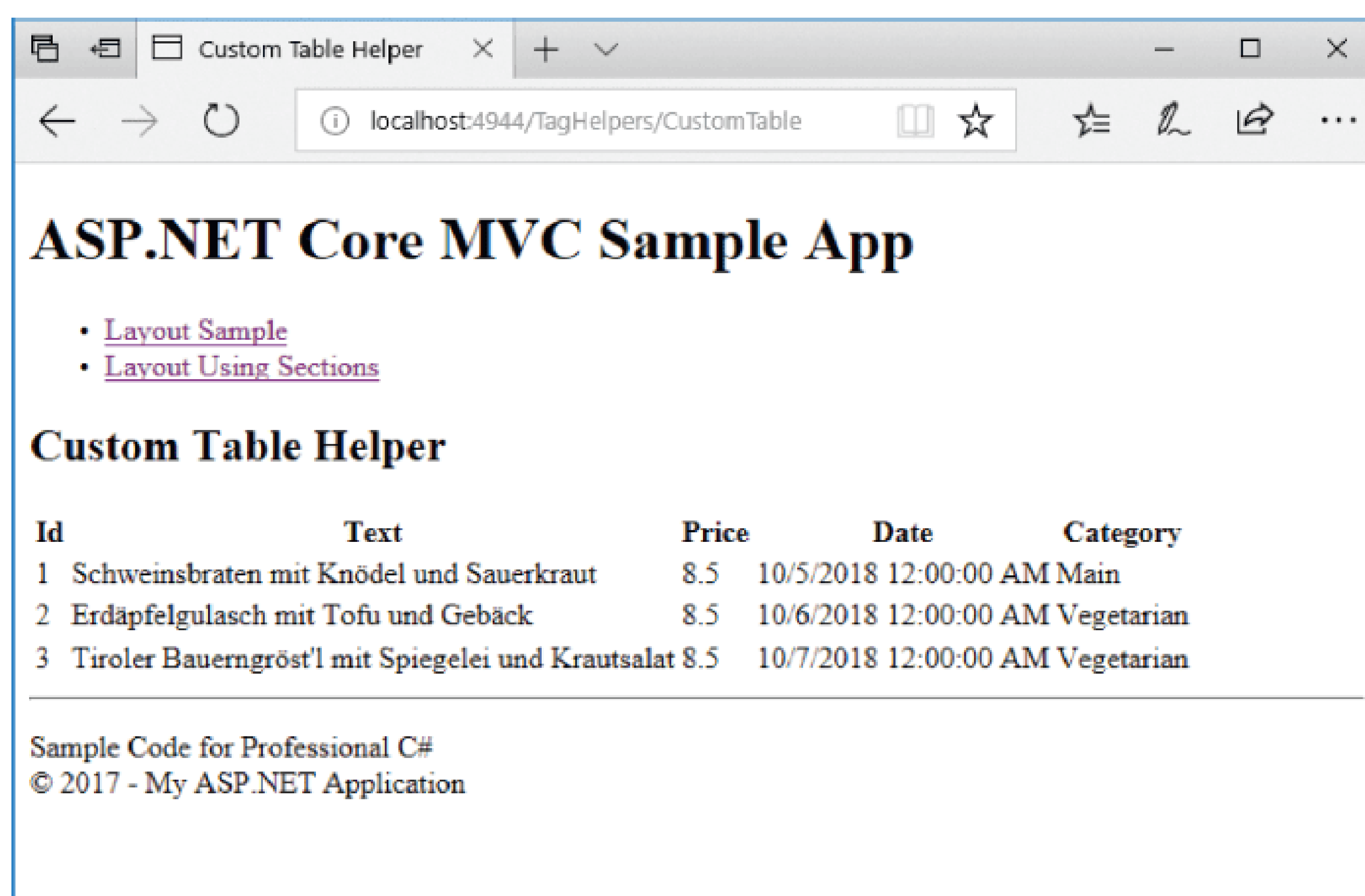



图 31-14

## 31.8　实现动作过滤器

ASP.NET Core MVC 在很多方面都可以扩展。可以实现控制器工厂，以搜索和实例化控制器(接口 IControllerFactory)。控制器实现了 IController 接口。使用 IActionInvoker 接口可以找出控制器中的动作方法。使用派生自 ActionMethodSelectorAttribute 的特性类可以定义允许的 HTTP 方法。通过实现 IModelBinder 接口，可以定制将 HTTP 请求映射到参数的模型绑定器。在 31.5.1 节"模型绑定器"中，使用过 FormCollectionModelBinder 类型。有实现了 IViewEngine 接口的不同视图引擎可供使用。在本章中，使用了 Razor 视图引擎。使用 HTML

Helper、Tag Helper 和动作过滤器也可以实现自定义。大多数可以扩展的地方都不在本书讨论范围内，但是由于很可能需要实现或使用动作过滤器，所以下面就加以讨论。

在动作执行之前和之后，都会调用动作过滤器。使用特性可把它们分配给控制器或控制器的动作方法。通过创建派生自基类 ActionFilterAttribute 的类，可以实现动作过滤器。在这个类中，可以重写基类成员 OnActionExecuting、OnActionExecuted、OnResultExecuting 和 OnResultExecuted。OnActionExecuting 在动作方法调用之前调用，OnActionExecuted 在动作方法完成之后调用。之后，在返回结果前，调用 OnResultExecuting 方法，最后调用 OnResultExecuted 方法。

在这些方法内，可以访问 Request 对象来检索调用者信息。然后根据浏览器决定执行某些操作、访问路由信息、动态修改视图结果等。下面的代码段访问路由信息中的变量 language。为把此变量添加到路由中，可以把路由修改为如 31.2 节“定义路由”所示。用路由信息添加 language 变量后，可以使用 RouteData.Values 访问 URL 中提供的值，如下面的代码段所示。可以根据得到的值，为用户修改区域性：

```
public class LanguageAttribute : ActionFilterAttribute
{
  private string _language = null;
  public override void OnActionExecuting(ActionExecutingContext filterContext)
  {
    _language = filterContext.RouteData.Values["language"] == null ?
      null : filterContext.RouteData.Values["language"].ToString();
    //...
  }

  public override void OnResultExecuting(ResultExecutingContext filterContext)
  {
  }
}
```

**注意：**
第 27 章讨论了全球化和本地化、区域性设置及其他区域信息。

使用创建的动作过滤器特性类，可以把该特性应用到一个控制器，如下面的代码段所示。对类应用特性后，在调用每个动作方法时，都会调用特性类的成员。另外，也可以把特性应用到一个动作方法，此时只有调用该动作方法时才会调用特性类的成员。

```
[Language]
public class HomeController : Controller
{
```

ActionFilterAttribute 实现了几个接口：IActionFilter、IAsyncActionFilter、IResultFilter、IAsyncResultFilter、IFilter 和 IOrderedFilter。

ASP.NET Core MVC 包含一些预定义的动作过滤器，例如需要 HTTPS、授权调用程序、处理错误或缓存数据的过滤器。

使用特性 Authorize 的内容参见本章后面的 31.10 节“实现身份验证和授权”。

## 31.9　创建数据驱动的应用程序

在讨论完 ASP.NET Core MVC 的基础知识后，创建一个使用 Entity Framework Core 的数据驱动的应用程序。该应用程序使用了 ASP.NET Core MVC 提供的功能和数据访问功能。

**注意：**
第 26 章详细讨论了 Entity Framework Core。

示例应用程序 MenuPlanner 用于维护数据库中存储的饭店菜单条目。数据库条目的维护只应该由经过身份验证的账户完成。但是，未经身份验证的用户应该能够浏览菜单。

这个项目首先选择 ASP.NET Core Web Application 模板。对于身份验证，选择默认选项 Individual User Accounts 和 Store User Accounts In-App。这个项目模板给 ASP.NET Core MVC 和控制器添加了几个文件夹，包括 HomeController 和 AccountController。另外还添加了几个脚本库。

### 31.9.1 定义模型

首先在 Models 目录中定义一个模型。该模型使用 Entity Framework Core(EF Core)创建。MenuCard 类型定义了一些属性和与一组菜单的关系(代码文件 MenuPlanner/Models/MenuCard.cs)：

```
public class MenuCard
{
  public int Id { get; set; }

  [MaxLength(50)]
  public string Name { get; set; }
  public bool Active { get; set; }
  public int Order { get; set; }
  public virtual List<Menu> Menus { get; set; }
}
```

在 MenuCard 中引用的菜单类型由 Menu 类定义(代码文件 MenuPlanner/Models/Menu.cs)：

```
public class Menu
{
  public int Id { get; set; }
  public string Text { get; set; }
  public decimal Price { get; set; }
  public bool Active { get; set; }
  public int Order { get; set; }
  public string Type { get; set; }
  public DateTime Day { get; set; }
  public int MenuCardId { get; set; }
  public virtual MenuCard MenuCard { get; set; }
}
```

数据库连接以及 Menu 和 MenuCard 类型的设置由 MenuCardsContext 管理。上下文使用 ModelBuilder 指定 Menu 类型的 Text 属性不能是 null，其最大长度是 50(代码文件 MenuPlanner/Models/MenuCardsContext.cs)：

```
public class MenuCardsContext : DbContext
{
  public MenuCardsContext(DbContextOptions<MenuCardsContext> options)
    : base(options)
  {
  }

  public DbSet<Menu> Menus { get; set; }
  public DbSet<MenuCard> MenuCards { get; set; }

  protected override void OnModelCreating(ModelBuilder modelBuilder)
  {
    modelBuilder.Entity<Menu>().Property(p => p.Text)
      .HasMaxLength(50).IsRequired();
    base.OnModelCreating(modelBuilder);
  }
}
```

Web 应用程序的启动代码定义了 MenuCardsContext，用作数据上下文，从配置文件中读取连接字符串(代码文件 MenuPlanner/Startup.cs)：

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(
      Configuration.GetConnectionString("DefaultConnection")));

  services.AddDbContext<MenuCardsContext>(options =>
    options.UseSqlServer(
      Configuration.GetConnectionString("DefaultConnection")));

  //...
}
```

在配置文件中，添加 DefaultConnection 连接字符串。这个连接字符串引用 Visual Studio 2017 附带的 SQL 实例。当然，也可以改变它，把这个连接字符串添加到 SQL Azure 中(代码文件 MenuPlanner/appsettings.json)：

```
{
  "ConnectionStrings": {
    "DefaultConnection":  "Server=(localdb)\\mssqllocaldb;Database=MenuPlanner;
      Trusted_Connection=True;MultipleActiveResultSets=true"
  },
  //...
}
```

## 31.9.2　创建数据库

可以使用 EF Core 命令创建代码来创建数据库。在命令行提示符中，使用.NET Core Command Line (CLI)和 ef 命令创建代码，来自动创建数据库。使用命令提示符时，必须把当前文件夹设置为 project 文件所在的目录：

```
>dotnet ef migrations add InitMenuCards --context MenuCardsContext
```

**注意：**
dotnet ef 工具扩展参见第 26 章。

因为这个项目定义了多个数据上下文(MenuCardsContext 和 ApplicationDbContext)，所以需要用--context 选项指定数据上下文。ef 命令在项目结构中创建一个 Migrations 文件夹，InitMenuCards 类使用 Up 方法来创建数据库表，使用 Down 方法再次删除更改(代码文件 MenuPlanner/Migrations/[date]InitMenuCards.cs)：

```
public partial class InitMenuCards : Migration
{
  public override void Up(MigrationBuilder migrationBuilder)
  {
    migrationBuilder.CreateTable(
      name: "MenuCards",
      columns: table => new
      {
        Id = table.Column<int>(nullable: false)
          .Annotation("SqlServer:ValueGenerationStrategy",
           SqlServerValueGenerationStrategy.IdentityColumn),
        Active = table.Column<bool>(nullable: false),
        Name = table.Column<string>(maxLength: 50, nullable: true),
        Order = table.Column<int>(nullable: false)
      },
      constraints: table =>
      {
        table.PrimaryKey("PK_MenuCards", x => x.Id);
      });

    migrationBuilder.CreateTable(
      name: "Menus",
      columns: table => new
      {
        Id = table.Column<int>(nullable: false)
          .Annotation("SqlServer:ValueGenerationStrategy",
           SqlServerValueGenerationStrategy.IdentityColumn),
        Active = table.Column<bool>(nullable: false),
        Day = table.Column<DateTime>(nullable: false),
        MenuCardId = table.Column<int>(nullable: false),
        Order = table.Column<int>(nullable: false),
        Price = table.Column<decimal>(nullable: false),
        Text = table.Column<string>(maxLength: 50, nullable: false),
        Type = table.Column<string>(nullable: true)
      },
      constraints: table =>
      {
        table.PrimaryKey("PK_Menus", x => x.Id);
        table.ForeignKey(
          name: "FK_Menus_MenuCards_MenuCardId",
          column: x => x.MenuCardId,
          principalTable: "MenuCards",
          principalColumn: "Id",
```

```
        onDelete: ReferentialAction.Cascade);
      });
  }

  public override void Down(MigrationBuilder migration)
  {
    migration.DropTable("Menus");
    migration.DropTable("MenuCards");
  }
}
```

现在只需要一些代码来启动迁移过程，用最初的样本数据填充数据库。MenuCardDatabaseInitializer 在 Database 属性返回的 DatabaseFacade 对象上调用扩展方法 MigrateAsync，应用迁移过程。这又反过来检查与连接字符串关联的数据库版本是否与迁移指定的数据库相同。如果版本不同，就需要调用 Up 方法得到相同的版本。此外，创建一些MenuCard对象，存储在数据库中(代码文件 MenuPlanner/Models/MenuCardDatabaseInitializer.cs)：

```
using Microsoft.EntityFrameworkCore;
using System.Threading.Tasks;

namespace MenuPlanner.Models
{
  public class MenuCardDatabaseInitializer
  {
    private static bool s_databaseChecked = false;
    private readonly MenuCardsContext _context;

    public MenuCardDatabaseInitializer(MenuCardsContext context) =>
      _context = context;

    public async Task CreateAndSeedDatabaseAsync()
    {
      if (!s_databaseChecked)
      {
        s_databaseChecked = true;
        await _context.Database.MigrateAsync();
        if (await _context.MenuCards.CountAsync() == 0)
        {
          _context.MenuCards.Add(
            new MenuCard { Name = "Breakfast", Active = true, Order = 1 });
          _context.MenuCards.Add(
            new MenuCard { Name = "Vegetarian", Active = true, Order = 2 });
          _context.MenuCards.Add(
            new MenuCard { Name = "Steaks", Active = true, Order = 3 });
        }
        await _context.SaveChangesAsync();
      }
    }
  }
}
```

有了数据库和模型，就可以创建服务了。

### 31.9.3　创建服务

在创建服务之前，创建了接口 IMenuCardsService，它定义了服务所需的所有方法(代码文件 MenuPlanner/Services/IMenuCardsService.cs)：

```
using MenuPlanner.Models;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace MenuPlanner.Services
{
  public interface IMenuCardsService
  {
    Task AddMenuAsync(Menu menu);
    Task DeleteMenuAsync(int id);
    Task<Menu> GetMenuByIdAsync(int id);
    Task<IEnumerable<Menu>> GetMenusAsync();
    Task<IEnumerable<MenuCard>> GetMenuCardsAsync();
    Task UpdateMenuAsync(Menu menu);
  }
}
```

服务类 MenuCardsService 实现了返回菜单和菜单卡的方法，并创建、更新和删除菜单(代码文件 MenuPlanner/Services/MenuCardsService.cs)：

```
using MenuPlanner.Models;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace MenuPlanner.Services
{
  public class MenuCardsService : IMenuCardsService
  {
    private readonly MenuCardsContext _menuCardsContext;
    public MenuCardsService(MenuCardsContext menuCardsContext) =>
      _menuCardsContext = menuCardsContext;

    public async Task<IEnumerable<Menu>> GetMenusAsync()
    {
      await EnsureDatabaseCreatedAsync();
      var menus = _menuCardsContext.Menus.Include(m => m.MenuCard);
      return await menus.ToArrayAsync();
    }

    public async Task<IEnumerable<MenuCard>> GetMenuCardsAsync()
    {
      await EnsureDatabaseCreatedAsync();
      var menuCards = _menuCardsContext.MenuCards;
      return await menuCards.ToArrayAsync();
    }

    public async Task<Menu> GetMenuByIdAsync(int id) =>
      await _menuCardsContext.Menus.SingleOrDefaultAsync(m => m.Id == id);

    public async Task AddMenuAsync(Menu menu)
    {
      await _menuCardsContext.Menus.AddAsync(menu);
      await _menuCardsContext.SaveChangesAsync();
    }

    public async Task UpdateMenuAsync(Menu menu)
    {
      _menuCardsContext.Menus.Update(menu);
      await _menuCardsContext.SaveChangesAsync();
    }

    public async Task DeleteMenuAsync(int id)
    {
      Menu menu = await _menuCardsContext.Menus.SingleAsync(m => m.Id == id);
      _menuCardsContext.Menus.Remove(menu);
      await _menuCardsContext.SaveChangesAsync();
    }

    private async Task EnsureDatabaseCreatedAsync()
    {
      var init = new MenuCardDatabaseInitializer(_menuCardsContext);
      await init.CreateAndSeedDatabaseAsync();
    }
  }
}
```

为了使服务可用于依赖注入，使用 AddScoped 方法在服务集合中注册服务(代码文件 MenuPlanner/Startup.cs)：

```
public void ConfigureServices(IServiceCollection services)
{
  //...
  services.AddScoped<IMenuCardsService, MenuCardsService>();
  //...
}
```

### 31.9.4 创建控制器

ASP.NET Core MVC 提供搭建功能来创建控制器，以直接访问数据库。为此，可以在 Solution Explorer 中选择 Controllers 文件夹，并从上下文菜单中选择 Add｜Controller。打开 Add Scaffold 对话框。在该对话框中，可以使用 Entity Framework 选择 MVC Controller 视图。单击 Add 按钮，打开 Add MVC Controller 对话框，如图 31-15 所示。使用此对话框，可以选择 Menu 模型类和 Entity Framework 数据上下文 MenuCardsContext，配置为生成视图，给控制器指定一个名称。用视图创建控制器，查看生成的代码，包括视图。

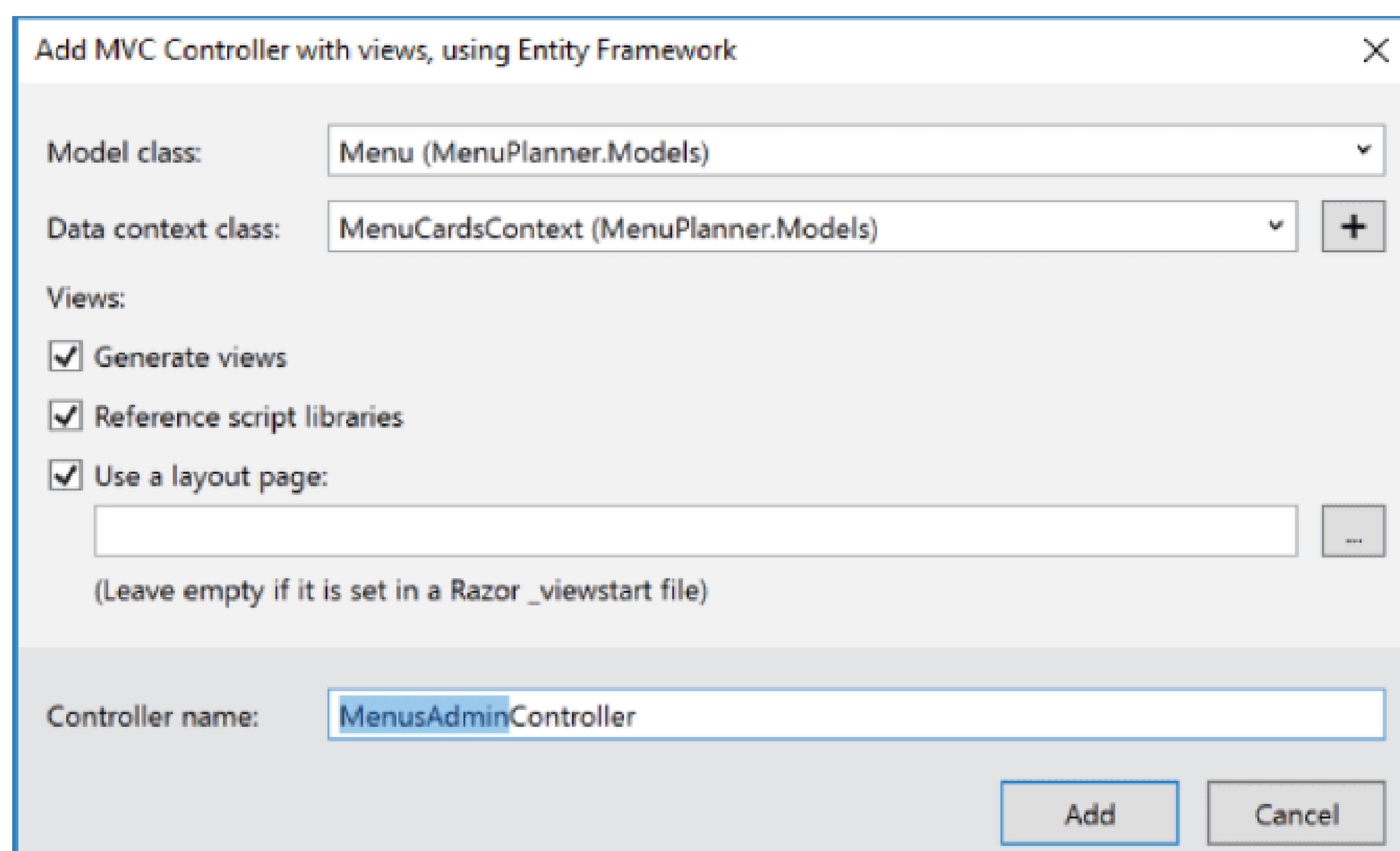



图 31-15

图书示例不直接在控制器中使用数据上下文，而是把一个服务放在其中。这样做提供了更多的灵活性。可以在不同的控制器中使用服务，还可以在服务中使用服务，如 ASP.NET Core Web API。

**注意：**
ASP.NET Core Web API 参见第 32 章。

在下面的示例代码中，ASP.NET Core MVC 控制器通过构造函数注入来注入菜单卡服务(代码文件 MenuPlanner/Controllers/MenusAdminController.cs)：

```
public class MenusAdminController : Controller
{
  private readonly IMenuCardsService _service;
  public MenusAdminController(IMenuCardsService service) =>
    _service = service;
  //...
}
```

只有当控制器通过 URL 来引用而没有传递动作方法时，才默认调用 Index 方法。这里，会创建数据库中所有的 Menu 项，并传递到 Index 视图。Details 方法传递在服务中找到的菜单，返回 Details 视图。注意错误处理。在没有把 ID 传递给 Details 方法时，如果在数据库中没有找到菜单，使用 NotFound 方法返回 HTTP Not Found 错误(404 错误响应)。这个方法在控制器的一个基类 ControllerBase 中定义(代码文件 MenuPlanner/Controllers/MenusAdminController.cs)：

```
public async Task<IActionResult> Index() =>
  View(await _service.GetMenusAsync());

public async Task<IActionResult> Details(int? id = 0)
{
  if (id == null)
  {
    return NotFound();
  }
```

```
    Menu menu = await _service.GetMenuByIdAsync(id.Value);
    if (menu == null)
    {
      return NotFound();
    }
    return View(menu);
  }
```

用户创建新菜单时，在收到客户端的 HTTP GET 请求后，会调用第一个 Create 方法。在这个方法中，把 ViewBag 信息传递给视图。这个 ViewBag 包含 SelectList 中菜单卡的信息。SelectList 允许用户选择一项。因为 MenuCard 集合被传递给 SelectList，所以用户可以选择一个带有新建菜单的菜单卡(代码文件 MenuPlanner/Controllers/MenusAdminController.cs)：

```
public async Task<IActionResult> Create()
{
  IEnumerable<MenuCard> cards = await _service.GetMenuCardsAsync();
  ViewBag.MenuCardId = new SelectList(cards, "Id", "Name");
  return View();
}
```

在用户填写表单并把带有新菜单的表单提交到服务器时，在 HTTP POST 请求中调用第二个 Create 方法。这个方法使用模型绑定，把表单数据传递给 Menu 对象，并将 Menu 对象添加到数据上下文中，向数据库写入新创建的菜单(代码文件 MenuPlanner/Controllers/MenusAdminController.cs)：

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Create(
  [Bind("Id","MenuCardId", "Text", "Price", "Active", "Order", "Type", "Day")]
    Menu menu)
{
  if (ModelState.IsValid)
  {
    await _service.AddMenuAsync(menu);
    return RedirectToAction("Index");
  }
  IEnumerable<MenuCard> cards = await _service.GetMenuCardsAsync();
  ViewBag.MenuCards = new SelectList(cards, "Id", "Name", menu.MenuCardId);
  return View(menu);
}
```

**注意：**
第 24 章解释了特性 ValidateAntiForgeryToken 的防伪令牌。

为了编辑菜单卡，定义了两种动作方法 Edit，一个用于 GET 请求，另一个用于 POST 请求。第一个 Edit 方法返回一个菜单项；第二个 Edit 方法在模型绑定成功后调用服务的 UpdateMenuAsync 方法：

```
public async Task<IActionResult> Edit(int? id)
{
  if (id == null)
  {
    return NotFound();
  }
  Menu menu = await _service.GetMenuByIdAsync(id.Value);
  if (menu == null)
  {
    return NotFound();
  }
  IEnumerable<MenuCard> cards = await _service.GetMenuCardsAsync();
  ViewBag.MenuCardId = new SelectList(cards, "Id", "Name", menu.MenuCardId);
  return View(menu);
}

[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id,
```

```
    [Bind("Id", "MenuCardId", "Text", "Price", "Order", "Type", "Day")]
      Menu menu)
  {
    if (id != menu.Id)
    {
      return NotFound();
    }

    if (ModelState.IsValid)
    {
      await _service.UpdateMenuAsync(menu);
      return RedirectToAction("Index");
    }
    IEnumerable<MenuCard> cards = await _service.GetMenuCardsAsync();
    ViewBag.MenuCardId = new SelectList(cards, "Id", "Name", menu.MenuCardId);
    return View(menu);
  }
```

控制器的实现的最后一部分包括 Delete 方法。因为这两个方法有相同的参数(这在 C#中是不可能的)，所以第二个方法的名称是 DeleteConfirmed。第二个方法可以在第一个 Delete 方法所在的 URL 链接中访问，但是它用 HTTP POST 访问，而不是用 ActionName 特性的 GET 访问。该方法调用服务的 DeleteMenuAsync 方法：

```
public async Task<IActionResult> Delete(int? id)
{
  if (id == null)
  {
    return NotFound();
  }
  Menu menu = await _service.GetMenuByIdAsync(id.Value);
  if (menu == null)
  {
    return NotFound();
  }
  return View(menu);
}

[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
  Menu menu = await _service.GetMenuByIdAsync(id);
  await _service.DeleteMenuAsync(menu.Id);
  return RedirectToAction("Index");
}
```

**注意：**

使用 Web API 时，删除资源通常是使用 HTTP DELETE 谓词完成的。有关 Web API 的信息参见第 32 章。对于访问页面的浏览器来说，情况并非如此。DeleteConfirmed 方法是使用 HTTP POST 谓词请求的(由 HttpPost 特性定义)。

### 31.9.5　创建视图

现在该创建视图了。视图在文件夹 Views/MenuAdmin 中创建。要创建视图，可以在 Solution Explorer 中选择 MenuAdmin 文件夹，并从上下文菜单中选择 Add | View。这将打开 Add MVC View 对话框，如图 31-16 所示。使用此对话框可以选择 List、Details、Create、Edit、Delete 模板，安排相应的 HTML 元素。在这个对话框中选择的 Model 类定义了视图基于的模型。

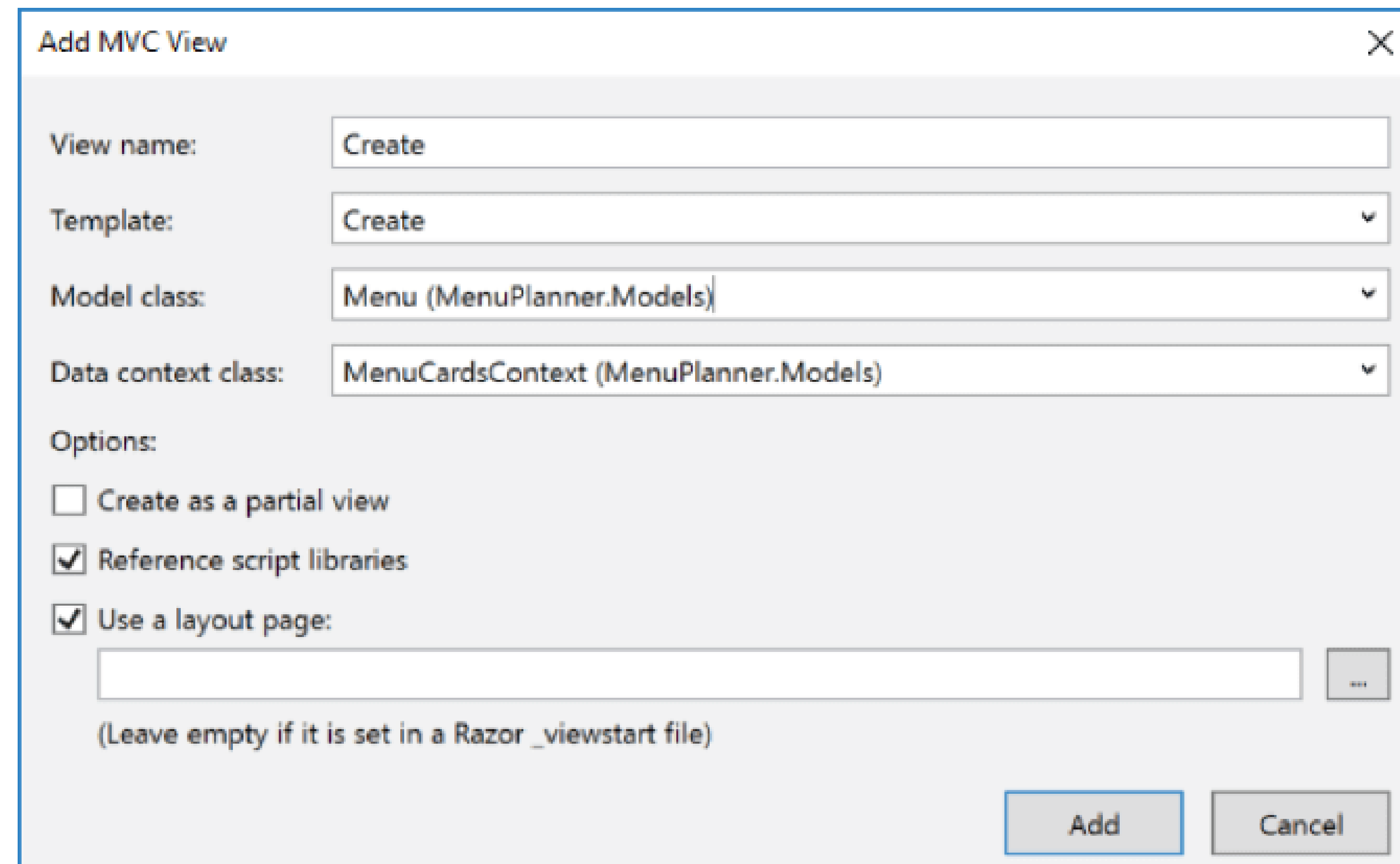



图 31-16

Index 视图定义了一个 HTML 表，它把 Menu 集合作为模型。对于表头元素，使用带有 HTML Helper DisplayNameFor 的 HTML 元素标签来访问用于显示的属性名称。为了显示项，菜单集合使用@foreach 迭代，每个属性值用输入元素的 HTML Helper 来访问。锚元素的 Tag Helper 为 Edit、Details 和 Delete 页面创建链接(代码文件 MenuPlanner/Views/MenuAdmin/Index.cshtml)：

```
@model IEnumerable<MenuPlanner.Models.Menu>
@{
   ViewData["Title"] = "Index";
}
<h2>Index</h2>
<p>
  <a asp-action="Create">Create New</a>
</p>
<table class="table">
  <thead>
    <tr>
      <th>
        @Html.DisplayNameFor(model => model.Text)
      </th>
      <th>
        @Html.DisplayNameFor(model => model.Price)
      </th>
      <th>
        @Html.DisplayNameFor(model => model.Active)
      </th>
      <th>
        @Html.DisplayNameFor(model => model.Order)
      </th>
      <th>
        @Html.DisplayNameFor(model => model.Type)
      </th>
      <th>
        @Html.DisplayNameFor(model => model.Day)
      </th>
      <th>
        @Html.DisplayNameFor(model => model.MenuCard)
      </th>
      <th></th>
    </tr>
  </thead>
  <tbody>
  @foreach (var item in Model) {
    <tr>
      <td>
        @Html.DisplayFor(modelItem => item.Text)
      </td>
      <td>
        @Html.DisplayFor(modelItem => item.Price)
      </td>
      <td>
        @Html.DisplayFor(modelItem => item.Active)
```

```
        </td>
        <td>
          @Html.DisplayFor(modelItem => item.Order)
        </td>
        <td>
          @Html.DisplayFor(modelItem => item.Type)
        </td>
        <td>
          @Html.DisplayFor(modelItem => item.Day)
        </td>
        <td>
          @Html.DisplayFor(modelItem => item.MenuCard.Id)
        </td>
        <td>
          <a asp-action="Edit" asp-route-id="@item.Id">Edit</a> |
          <a asp-action="Details" asp-route-id="@item.Id">Details</a> |
          <a asp-action="Delete" asp-route-id="@item.Id">Delete</a>
        </td>
      </tr>
    }
  </tbody>
</table>
```

在 MenuPlanner 项目中，MenuAdmin 控制器的第二个视图是 Create 视图。HTML 表单使用 asp-action Tag Helper 来引用控制器的 Create 动作方法。用 asp-controller Helper 引用控制器不是必要的，因为动作方法与视图位于相同的控制器中。表单内容使用标签和输入元素的 Tag Helper 建立。标签的 asp-for Helper 返回属性的名称；输入元素的 asp-for Helper 返回其值(代码文件 MenuPlanner/Views/MenuAdmin/Create.cshtml)：

```
@model MenuPlanner.Models.Menu
@{
    ViewData["Title"] = "Create";
}
<h2>Create</h2>
<h4>Menu</h4>
<hr />
<div class="row">
  <div class="col-md-4">
    <form asp-action="Create">
      <div asp-validation-summary="ModelOnly" class="text-danger"></div>
      <div class="form-group">
        <label asp-for="Text" class="control-label"></label>
        <input asp-for="Text" class="form-control" />
        <span asp-validation-for="Text" class="text-danger"></span>
      </div>
      <div class="form-group">
        <label asp-for="Price" class="control-label"></label>
        <input asp-for="Price" class="form-control" />
        <span asp-validation-for="Price" class="text-danger"></span>
      </div>
      <div class="form-group">
        <div class="checkbox">
          <label>
            <input asp-for="Active" />
            @Html.DisplayNameFor(model => model.Active)
          </label>
        </div>
      </div>
      <div class="form-group">
        <label asp-for="Order" class="control-label"></label>
        <input asp-for="Order" class="form-control" />
        <span asp-validation-for="Order" class="text-danger"></span>
      </div>
      <div class="form-group">
        <label asp-for="Type" class="control-label"></label>
        <input asp-for="Type" class="form-control" />
        <span asp-validation-for="Type" class="text-danger"></span>
      </div>
      <div class="form-group">
        <label asp-for="Day" class="control-label"></label>
        <input asp-for="Day" class="form-control" />
        <span asp-validation-for="Day" class="text-danger"></span>
      </div>
      <div class="form-group">
        <label asp-for="MenuCardId" class="control-label"></label>
        <select asp-for="MenuCardId" class ="form-control"
          asp-items="ViewBag.MenuCardId"></select>
```

```
      </div>
      <div class="form-group">
        <input type="submit" value="Create" class="btn btn-default" />
      </div>
    </form>
  </div>
</div>

<div>
  <a asp-action="Index">Back to List</a>
</div>

@section Scripts {
  @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

其他视图的创建与前面所示的视图类似，因此这里不作介绍。

现在可以使用应用程序在现有的菜单卡中添加和编辑菜单。

# 31.10　实现身份验证和授权

身份验证和授权是 Web 应用程序的重要方面。如果网站或其中的一部分不应公开，那么用户就必须获得授权。对于用户的身份验证，在创建 ASP.NET Core Web 应用程序时可以使用不同的选项(参见图 31-17)：No Authentication、Individual User Accounts、Work or School Accounts 和 Windows Authentication。

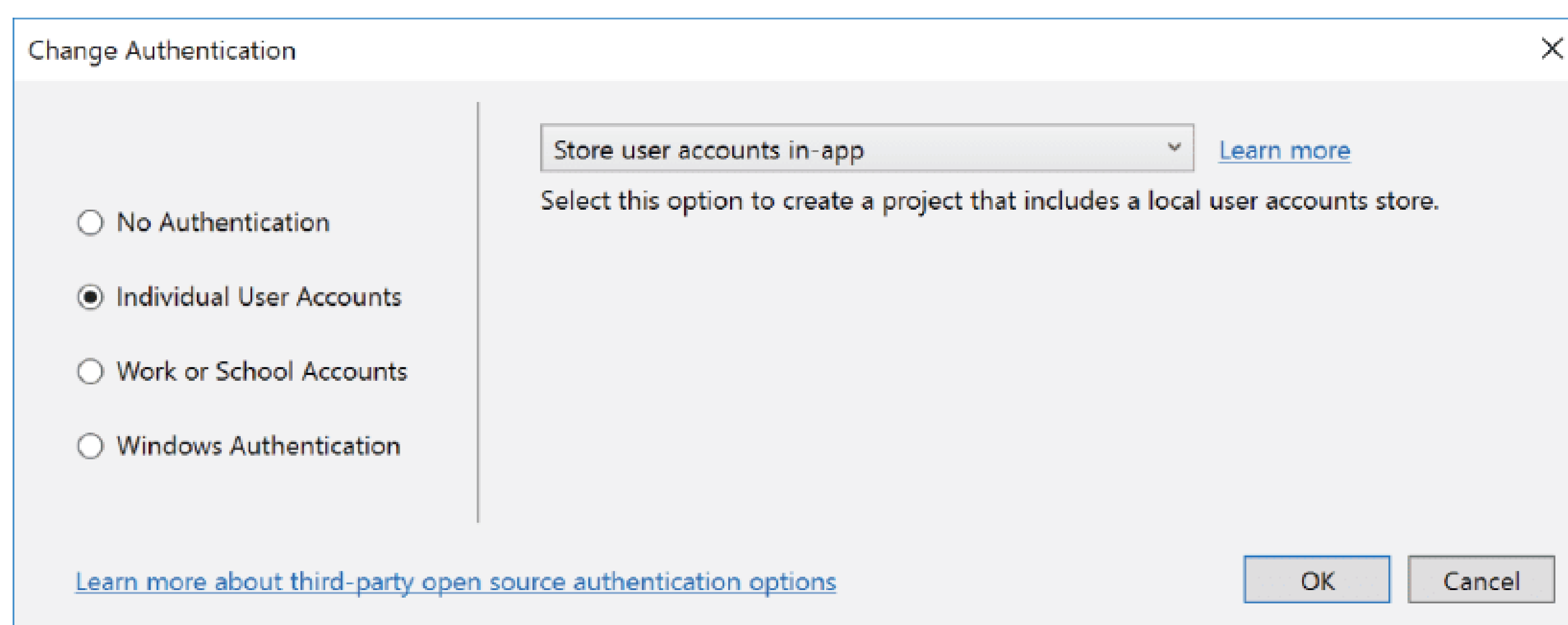



图 31-17

对于 Work or School Accounts，可以从云中选择 Active Directory，进行身份验证。

使用 Individual User Accounts 时，可以在 SQL Server 数据库中存储用户，或者使用 Azure Active Directory B2C。本章会讨论这两个选项。用户可以注册和登录，也可以使用 Facebook、Twitter、Google 和 Microsoft 中现有的账户。

## 31.10.1　存储和检索用户信息

为了管理用户，需要把用户信息添加到库中。IdentityUser 类型(名称空间 Microsoft.AspNet.Identity.EntityFramework)定义了一个名称，列出了角色、登录名和声明。用来创建 MenuPlanner 应用程序的 Visual Studio 模板创建了一些明显的代码来保存用户：类 ApplicationUser 是项目的一部分，派生自基类 IdentityUser(名称空间 Microsoft.AspNet.Identity.EntityFramework)。ApplicationUser 默认为空，但是可以添加需要的用户信息，这些信息存储在数据库中(代码文件 MenuPlanner/Models/IdentityModels.cs)：

```
public class ApplicationUser : IdentityUser
{
}
```

数据库的连接通过 IdentityDbContext<TUser>类型建立。这是一个泛型类，派生于 DbContext，因此使用了 Entity Framework Core。IdentityDbContext<TUser>类型定义了 IDbSet<TEntity>类型的 Roles 和 Users 属性。

IDbSet<TEntity>类型定义了到数据库表的映射。为了方便起见，创建 ApplicationDbContext，把 ApplicationUser 类型定义为 IdentityDbContext 类的泛型类型(代码文件 MenuPlanner/Data/ApplicationDbContext.cs)：

```
public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
{
  protected override void OnModelCreating(ModelBuilder builder)
  {
    base.OnModelCreating(builder);
  }
}
```

### 31.10.2 启动身份系统

数据库的连接通过启动代码中的依赖注入服务集合来注册。类似于前面创建的 MenuCardsContext，ApplicationDbContext 被配置为使用 SQL Server 和 config 文件中的连接字符串。身份服务本身使用扩展方法 AddIdentity 注册。AddIdentity 方法映射身份服务所使用的用户和角色类的类型。类 ApplicationUser 是前面提到的源自 IdentityUser 的类；IdentityRole 是基于字符串的角色类，派生自 IdentityRole<string>。AddIdentity 方法的重载版本允许的配置身份系统的方式有双因素身份验证；电子邮件令牌提供程序；用户选项，如需要唯一的电子邮件；或者正则表达式，要求用户名匹配。AddIdentity 返回一个 IdentityBuilder，允许对身份系统进行额外的配置，如使用的实体框架上下文(AddEntityFrameworkStores)和令牌提供程序(AddDefaultTokenProviders)。可以添加的其他提供程序用于错误、密码验证器、角色管理器、用户管理器和用户验证器(代码文件 MenuPlanner/Startup.cs)：

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(
      Configuration.GetConnectionString("DefaultConnection")));

  services.AddDbContext<MenuCardsContext>(options =>
    options.UseSqlServer(
      Configuration.GetConnectionString("DefaultConnection")));
  services.AddScoped<IMenuCardsService, MenuCardsService>();

  services.AddIdentity<ApplicationUser, IdentityRole>()
    .AddEntityFrameworkStores<ApplicationDbContext>()
    .AddDefaultTokenProviders();

  services.AddTransient<IEmailSender, EmailSender>();

  services.AddMvc();
}
```

### 31.10.3 执行用户注册

现在进入为注册用户而生成的代码。功能的核心在 AccountController 类中。控制器类应用了 Authorize 特性，它将所有的动作方法限制为通过身份验证的用户。构造函数接收一个用户管理器、登录管理器和通过依赖注入的数据库上下文。电子邮件和 SMS 发送方用于双因素身份验证。如果没有实现生成代码中的空 AuthMessageSender 类，就可以删除 IEmailSender 的注入(代码文件 MenuPlanner/Controllers/AccountController.cs)：

```
[Authorize]
[Route("[controller]/[action]")]
public class AccountController : Controller
{
  private readonly UserManager<ApplicationUser> _userManager;
  private readonly SignInManager<ApplicationUser> _signInManager;
  private readonly IEmailSender _emailSender;
  private readonly ILogger _logger;

  public AccountController(
    UserManager<ApplicationUser> userManager,
    SignInManager<ApplicationUser> signInManager,
    IEmailSender emailSender,
    ILogger<AccountController> logger)
  {
```

```
    _userManager = userManager;
    _signInManager = signInManager;
    _emailSender = emailSender;
    _logger = logger;
}
```

要注册用户，应定义 RegisterViewModel。这个模型定义了用户在注册时需要输入什么数据。在生成的代码中，这个模型只需要电子邮件、密码和确认密码(必须与密码相同)。如果想获得更多的用户信息，可以根据需要添加属性(代码文件 MenuPlanner/Models/AccountViewModels.cs)：

```
public class RegisterViewModel
{
  [Required]
  [EmailAddress]
  [Display(Name = "Email")]
  public string Email { get; set; }

  [Required]
  [StringLength(100, ErrorMessage =
    "The {0} must be at least {2} characters long.", MinimumLength = 6)]
  [DataType(DataType.Password)]
  [Display(Name = "Password")]
  public string Password { get; set; }

  [DataType(DataType.Password)]
  [Display(Name = "Confirm password")]
  [Compare("Password", ErrorMessage =
    "The password and confirmation password do not match.")]
  public string ConfirmPassword { get; set; }
}
```

用户注册对于未经过身份验证的用户也必须可用。这就是为什么 AllowAnonymous 特性应用于 AccountController 的 Register 方法的原因。这会否决这些方法的 Authorize 特性。Register 方法的 HTTP POST 变体接收 RegisterViewModel 对象，通过调用方法_userManager.CreateAsync 把 ApplicationUser 写入数据库。用户成功创建后，使用电子邮件提供程序向用户发送电子邮件，通过_signInManager.SignInAsync 登录(代码文件 MenuPlanner/Controllers/AccountController.cs)：

```
[HttpGet]
[AllowAnonymous]
public IActionResult Register(string returnUrl = null)
{
  ViewData["ReturnUrl"] = returnUrl;
  return View();
}

[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Register(RegisterViewModel model,
  string returnUrl = null)
{
  ViewData["ReturnUrl"] = returnUrl;
  if (ModelState.IsValid)
  {
    var user = new ApplicationUser
    {
      UserName = model.Email,
      Email = model.Email
    };
    var result = await _userManager.CreateAsync(user, model.Password);
    if (result.Succeeded)
    {
      _logger.LogInformation("User created a new account with password.");

      var code = await _userManager.GenerateEmailConfirmationTokenAsync(user);
      var callbackUrl = Url.EmailConfirmationLink(user.Id, code,
        Request.Scheme);
      await _emailSender.SendEmailConfirmationAsync(model.Email, callbackUrl);

      await _signInManager.SignInAsync(user, isPersistent: false);
      _logger.LogInformation("User created a new account with password.");
      return RedirectToLocal(returnUrl);
    }
    AddErrors(result);
```

```
    }
    // If we got this far, something failed, redisplay form
    return View(model);
}
```

现在视图(代码文件 MenuPlanner/Views/Account/Register.cshtml)只需要用户的信息。图 31-18 显示要求用户提供信息的对话框。

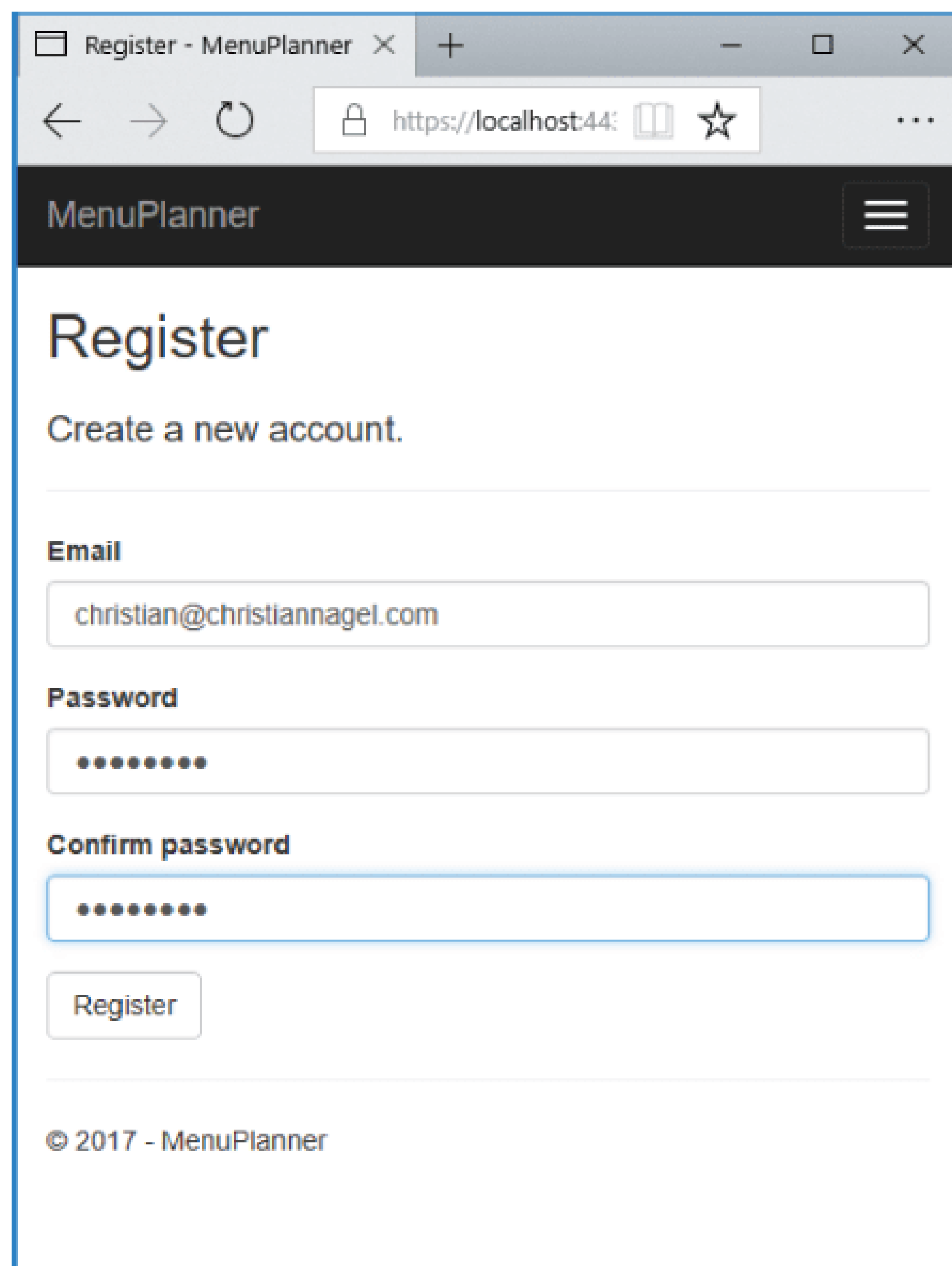



图 31-18

### 31.10.4 设置用户登录

当用户注册时，在注册成功后，会直接开始登录。LoginViewModel 模型定义了 UserName、Password 和 RememberMe 属性——用户登录时要求提供的信息。这个模型有一些注解与 HTML Helper 一起使用(代码文件 MenuPlanner/Models/AccountViewModels.cs)：

```
public class LoginViewModel
{
  [Required]
  [EmailAddress]
  public string Email { get; set; }

  [Required]
  [DataType(DataType.Password)]
  public string Password { get; set; }

  [Display(Name = "Remember me?")]
  public bool RememberMe { get; set; }
}
```

为了登录已注册的用户，需要调用 AccountController 的 Login 方法。用户输入登录信息后，就使用登录管理器通过 PasswordSignInAsync 验证登录信息。如果登录成功，用户就重定向到最初请求的页面。如果登录失败了，会返回同样的视图，再给用户提供一个选项，以正确输入用户名和密码(代码文件 MenuPlanner/Controllers/AccountController.cs)：

```
[HttpGet]
[AllowAnonymous]
public IActionResult Login(string returnUrl = null)
{
  Await HttpContext.SignOutAsync(IdentityConstants.ExternalScheme);
```

```
    ViewData["ReturnUrl"] = returnUrl;
    return View();
  }

  [HttpPost]
  [AllowAnonymous]
  [ValidateAntiForgeryToken]
  public async Task<IActionResult> Login(LoginViewModel model,
    string returnUrl = null)
  {
    ViewData["ReturnUrl"] = returnUrl;
    if (ModelState.IsValid)
    {
      var result = await _signInManager.PasswordSignInAsync(
        model.Email, model.Password, model.RememberMe, lockoutOnFailure: false);
      if (result.Succeeded)
      {
        _logger.LogInformation("User logged in.");
        return RedirectToLocal(returnUrl);
      }
      if (result.RequiresTwoFactor)
      {
        return RedirectToAction(nameof(LoginWith2fa),
          new { returnUrl, model.RememberMe });
      }
      if (result.IsLockedOut)
      {
        _logger.LogWarning("User account locked out.");
        return RedirectToAction(nameof(Lockout));
      }
      else
      {
        ModelState.AddModelError(string.Empty, "Invalid login attempt.");
        return View(model);
      }
    }
    return View(model);
  }
```

### 31.10.5　验证用户的身份

有了身份验证的基础设施，就很容易使用 Authorize 特性注解控制器或动作方法，要求用户进行身份验证。把这个特性应用到类上需要为类的每一个动作方法指定角色。如果不同的动作方法有不同的授权要求，Authorize 特性也可以应用于动作方法。使用这个特性，会验证调用者是否已经获得授权(检查授权 cookie)。如果调用者还没有获得授权，就返回一个 401 HTTP 状态代码，并重定向到登录动作。

应用特性 Authorize 时如果没有设置参数，那么就需要用户通过身份验证。为了拥有更多的控制，可以把角色赋予 Roles 属性，指定只有特定的用户角色才可以访问动作方法，如下面的代码段所示：

```
[Authorize(Roles="Menu Admins")]
public class MenuAdminController : Controller
{
```

还可以使用 Controller 基类的 User 属性访问用户信息，允许更动态地批准或拒绝用户。例如，根据传递的参数值，要求不同的角色。

**注意：**
*用户身份验证和其他安全信息参见第 24 章。*

### 31.10.6　使用 Azure Active Directory 对用户进行身份验证

现在最好不要在自己的数据库中保存用户名和密码。用户不喜欢记住另一个密码。用户可能在使用密码管理工具，因为用户不可能处理很多密码。这不是唯一的问题。使用本章前几节中使用过的 ASP.NET Core 标识系统来满足安全需求。密码是散列的，因此无法从数据库中读取。这只是一个从密码到散列的单向进程。仍然需要确保用户信息的安全性。

**注意：**
*最好将用户管理与应用程序放在单独的服务中。*

对于用户名和密码，可以使用用户已有的账户，如微软、谷歌、Facebook 和 Twitter 账户，并使用这些提供程序的令牌。这样用户就不需要处理额外的密码，而且所有的密码管理都不是应用程序的一部分。要实现这一点，需要使用所选择的提供程序创建应用程序，并相应地配置 ASP.NET Core 身份。使用这些提供程序中的一个或多个，只需要添加关于用户的附加信息，以及来自数据库中提供程序的用户标识符。

为了管理这个附加的用户信息，最佳实践是将其与应用程序分开。可以为这个功能编写一个服务应用程序，并在几个 Web 应用程序中使用它。可以使用开箱即用的服务，例如身份服务器(https://identityserver.io/)，并将其托管在自己的 Web 应用程序中，或者可以使用 PaaS 服务，如 Azure Active Directory。本节演示了 Azure AD B2C (Active Directory Business to Consumer)和 ASP.NET Core MVC 的用法。与 Azure AD 相反，Azure AD B2C 允许用户注册，并允许添加微软、谷歌、Facebook、Twitter 等其他提供程序。

### 1. 创建 Azure Active Directory B2C 租户

在 ASP.NET Core 中使用 Azure Active Directory B2C 时，首先需要创建 Azure Active Directory。这可以通过门户网站 https://portal.azure.com 来实现。我用域名 procsharp.onmicrosoft.com 创建了一个。需要创建一个不同的域名，因为这个域名已经不可用了。

要让 Web 应用程序访问这个 Azure AD B2C，需要创建一个应用程序，如图 31-19 所示。除了输入应用程序的名称之外，还需要选择 Web App 选项，并配置 Reply URI。在创建 ASP.NET Core Web 应用程序时，将得到 Reply URI。例如 https://localhost:44359/signin-oidc。端口在用户的系统上是不同的。仅为了测试目的，可以使用 localhost。对于产品，需要将其更改为产品服务器的 URL。



图 31-19

要使用户能够从另一个提供程序中输入新的用户名和密码，可以添加标识提供程序，如图 31-20 所示。对于所选择的每个身份提供程序，都需要配置客户端 ID 和客户端密钥。对于使用微软账户，可以在 https://apps.dev.microsoft.com/中注册一个应用程序来检索 ID 和密钥。

图 31-20

接下来需要一些策略。通过策略，可以指定注册用户需要什么信息，以及可以使用哪些提供程序。对于 ASP.NET Core Web 应用程序，至少需要注册或登录和密码重置策略。自定义注册或登录策略(参见图 31-21)时，要定义身份提供程序，以使用电子邮件、城市和国家等注册属性。用户需要在注册时提供这些信息。还可以指定应该将哪些信息放入应用程序声明中——在访问令牌中发送给应用程序的信息。在使用 Azure AD B2C 定义用户属性时，可以定义出现在这里的自定义字段。

图 31-21

配置 Azure AD B2C 租户后，可以构建 ASP.NET Core Web 应用程序。

### 2. 用 Azure AD B2C 创建 ASP.NET Core Web 应用程序

当创建 ASP.NET Core Web 应用程序(Model-View-Controller)时，需要使用 Connect to an existing user store in the cloud 选项，将身份验证设置更改为 Individual User Accounts(参见图 31-22)。在这个对话框中，还需要指定域名、应用程序 ID、回调路径和使用 Azure AD B2C 配置指定的策略。所有这些设置都保存到配置文件 appsettings.json 中，可以在以后修改它们。

图 31-22

生成的启动代码为 Startup 类中的身份验证注册所需的服务。调用 AddAuthentication 扩展方法来配置 cookie 和 OpenID 标准。对于 Azure AD B2C，扩展方法 AddAzureAdB2C 在 AzureAdB2CAuthenticationBuilderExtensions 类的项目 Extensions 文件夹中定义。这个方法采用设置中的 AzureAdB2C 部分来配置身份验证(代码文件 AzureADB2CSample/Startup.cs)：

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddAuthentication(sharedOptions =>
  {
    sharedOptions.DefaultScheme =
      CookieAuthenticationDefaults.AuthenticationScheme;
    sharedOptions.DefaultChallengeScheme =
      OpenIdConnectDefaults.AuthenticationScheme;
  })
  .AddAzureAdB2C(options => Configuration.Bind("AzureAdB2C", options))
  .AddCookie();

  services.AddMvc();
}
```

有了这个配置，就可以启动应用程序，然后单击 Sign-In 按钮，通过已配置的社交账户身份提供程序或使用用户名和密码登录，如图 31-23 所示。如果用户还没有注册，就可以通过单击 Sign Up Now 链接进行注册。此对话框根据策略请求用户的信息(参见图 31-24)。

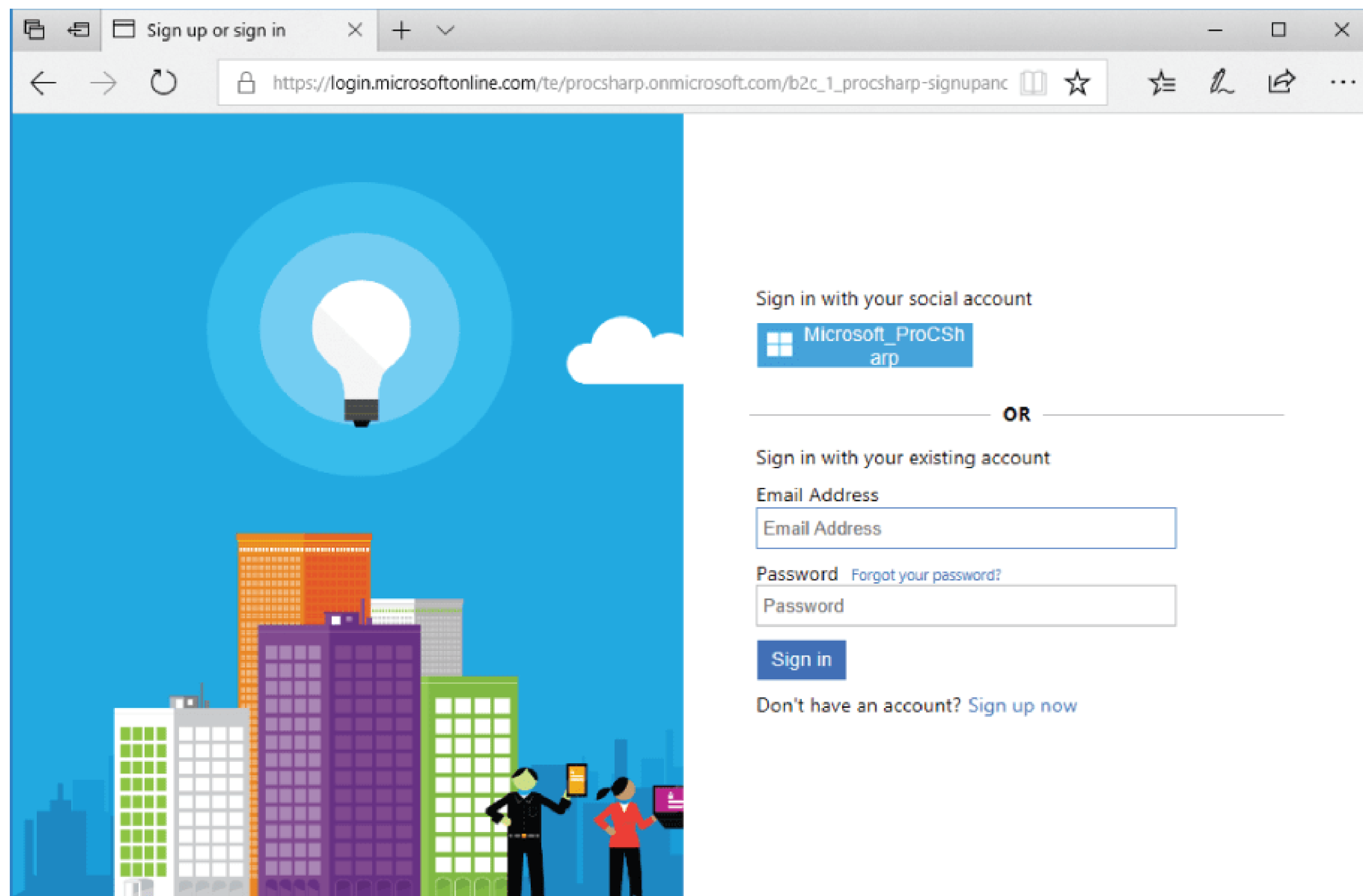



图 31-23

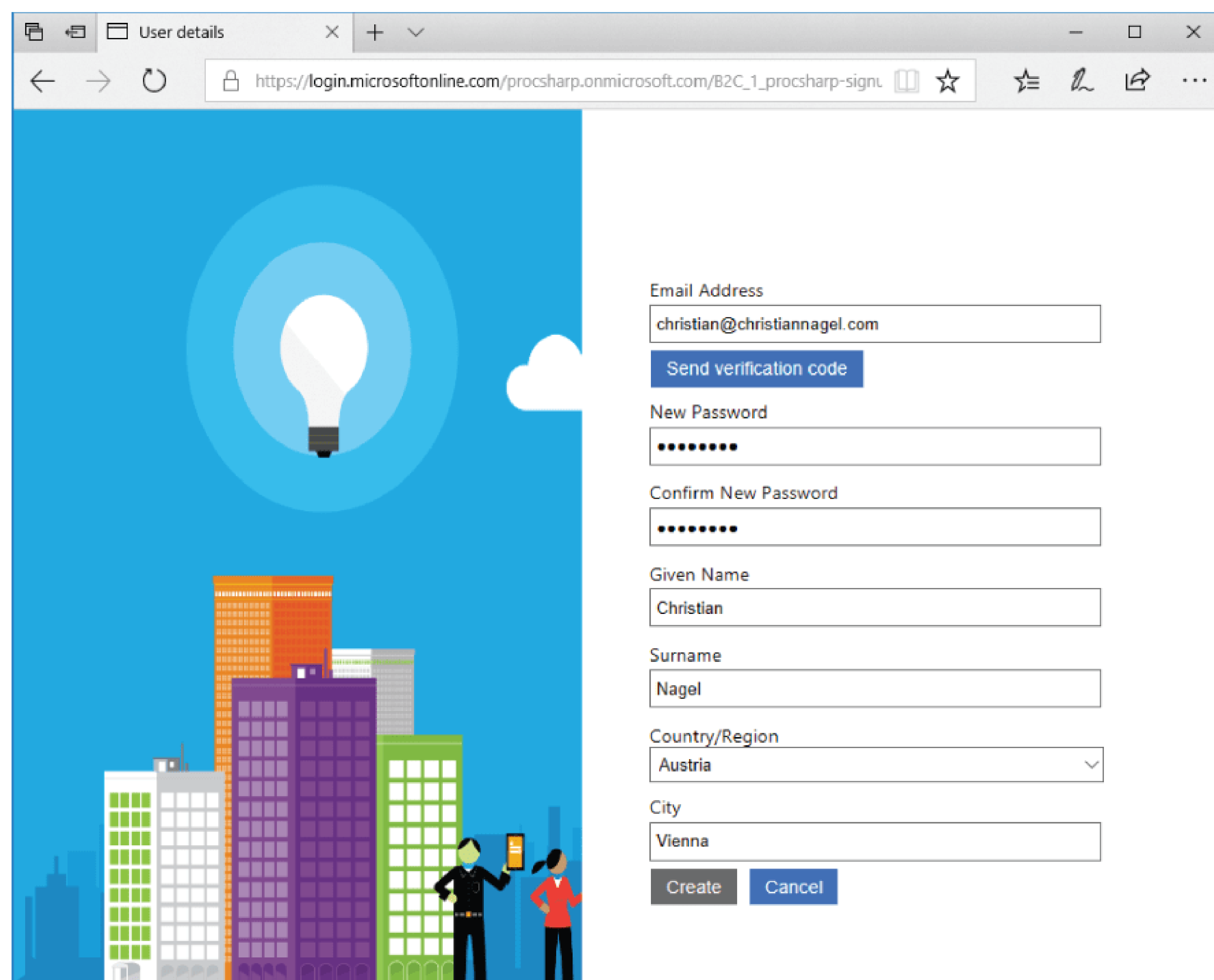



图 31-24

使用 Azure AD B2C，主要需要进行配置，并编写较少的代码来集成应用程序。这样可以更好地关注业务代码。

## 31.11 Razor 页面

ASP.NET Core 2.0 提供了一种创建 Web 应用程序的新技术：Razor 页面。与 MVC 模式相比，这种技术更容易理解，有时，控制器只是将请求转发到视图。在前面的示例中，已经使用了一些操作方法，它们仅仅调用 View 方法。如果创建一个带有脚本库的 Single Page Application (SPA)，比如 Angular，只要使用 Razor 页面就可以启动 Angular 页面。

使用 Razor 页面，就不需要分离模型-视图-控制器。仍然像使用 MVC 和视图一样编写 CSHTML 页面。使用 Razor 页面，可以直接在 CSHTML 页面中添加 C#代码——不仅仅是对视图使用的 Razor 语法。还可以使用与 CSHTML 文件直接相关的代码隐藏文件，并且可以与 C#代码进行一些分离。代码隐藏文件类似于其他技术(如 WPF、UWP 和 ASP.NET Web Forms)。

Razor 页面是基于 ASP.NET Core MVC 的。NuGet 包 Microsoft.AspNetCore.All 包括对 Razor 页面的支持。可以使用与视图相同的功能，如 Razor 语法、HTML Helper、Tag Helper、视图中的依赖注入等。不需要控制器。CSHTML 文件有一个@Page 指令，可以将 C#代码添加到页面的代码隐藏文件中。

因为 Razor 页面使用的特性与 ASP.NET Core MVC 中的视图相同，本节只讨论 ASP.NET Core MVC 和 Razor 页面之间的差异。

### 31.11.1 创建一个 Razor 页面项目

使用 Visual Studio 时，可以使用项目模板 Web Application with ASP.NET Core 2.0 创建一个 Razor Page 应用程序。示例项目 RazorPagesSample 就使用这个项目模板(参见图 31-25)。还可以从 Web Application (Model-View-Controller)模板开始，并向这个项目添加 Razor 页面。只需要将 Razor 页面文件添加到 Pages 文件夹中。将 Razor 页面与 MVC 混合起来很容易。

如果使用命令行，则使用如下命令创建 Razor 页面项目：

```
>dotnet new razor
```

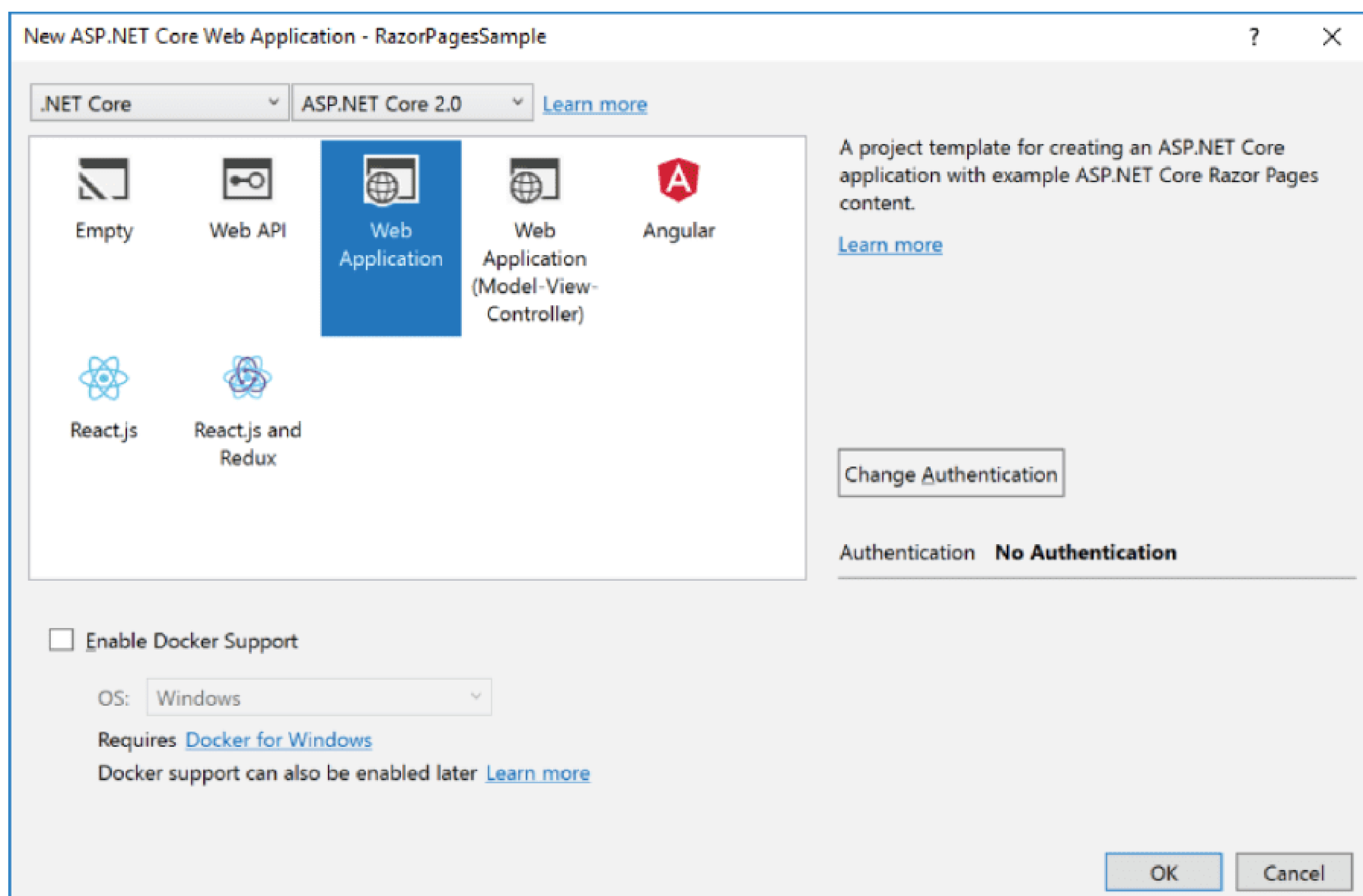



图 31-25

Startup 类的 ConfigureServices 方法与在 ASP.NET Core MVC 项目中的类似，包含对 AddMvc 的调用，以注

册 ASP.NET Core MVC 需要的所有服务(代码文件 RazorPagesSample/Startup.cs)：

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddMvc();
}
```

AddMvc 的调用包括 Razor 页面所需的服务。Razor 也可以通过使用 AddRazorOptions 和 AddRazorPageOptions 方法来配置。默认情况下，在 Pages 文件夹和子文件夹中搜索 Razor 页面。例如，如果访问 URL /Hello，就搜索页面 Pages/Hello.cshtml。使用 URL /Admin/User，会得到页面 Pages/Admin/User.cshtml。

如果没有找到这些页面，则在 views/Shared 文件夹中继续搜索。用 AddRazorOptions 方法设置属性 PageViewLocationFormats 时，可以改变这种行为。只需要将文件夹 Pages 更改为另一个文件夹，就可以使用 AddRazorPagesOptions 方法设置 RazorPagesOptions 的 RootDirectory 属性。

## 31.11.2　实现数据访问

示例应用程序将使用 EF Core 从数据库中读写。这与之前的内容类似，因此不需要专门解释。Book 和 BooksContext 类是在 Model 目录中创建的，以说明可以使用来自 ASP.NET Core MVC 和 Razor 页面中基于服务的代码。

下面代码片段中的 Book 类定义了要读写的数据的模型(代码文件 RazorPagesSample/Models/Book.cs)：

```
public class Book
{
  public int BookId { get; set; }

  [StringLength(50)]
  public string Title { get; set; }

  [StringLength(20)]
  public string Publisher { get; set; }
}
```

BooksContext 类将 Book 类型映射到 Books 表，并实现为与依赖注入一起使用(代码文件 RazorPagesSample/Models/BooksContext.cs)：

```
public class BooksContext : DbContext
{
  public BooksContext(DbContextOptions<BooksContext> options)
    : base(options) { }

  public DbSet<Book> Books { get; set; }
}
```

BooksContext 类现在使用 AddDbContext 扩展方法在依赖注入容器中注册(代码文件 RazorPagesSample/Startup.cs)：

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddMvc();
  services.AddDbContext<BooksContext>(options =>
    options.UseSqlServer(
      Configuration.GetConnectionString("BooksConnection")));
}
```

对于数据库管理，剩下的就是定义连接字符串(配置文件 RazorPagesSample/appsettings.json)：

```
{
  "ConnectionStrings": {
    "BooksConnection":  "server=(localdb)\\mssqllocaldb;database=RazorBooks;
trusted_connection=true"
  }
}
```

到目前为止，Razor 页面的代码与使用 ASP.NET Core MVC 中的控制器没有什么不同。Razor 页面最有趣的部分将在下一节中介绍。

### 31.11.3 使用内联代码

在创建项目并添加代码以访问数据库之后，可以使用 Visual Studio Item Template Razor Pages 来添加 Razor 页面。使用这个模板，可以在简单的 Razor 页面、使用 Entity Framework 的 Razor 页面和使用 Entity Framework 的多个 Razor 页面之间选择(参见图 31-26)。当对话框列出 Entity Framework 时，可以使用 EF Core 来创建这些页面。下面从一个简单的 Razor 页面开始。

在选择 Razor 页面之后，可以选择 Generate a PageModel class、Create as a Partial View 和 Use a Layout Page 选项(参见图 31-26)。虽然本章已经介绍了 ASP.NET Core MVC 中的部分视图和布局页面，但 PageModel 的选项是 Razor 页面新增的。选择 Generate a PageModel class 选项时，将生成代码隐藏文件；没有 PageModel，所有 C#代码都需要放在 CSHTML 文件中。使用示例代码，第一个创建的页面将是内联的，没有代码隐藏文件。

从浏览器中请求页面时，数据库中的所有图书都应该显示在一个表中。在 CSHTML 文件的顶部有一个 Page 指令。此指令将文件标记为 Razor 页面。因为 BooksContext 类是在依赖注入容器中注册的，所以可以使用@inject 直接注入页面中。(代码文件 RazorPagesSample/Pages/Inline.cshtml)：

```
@page
@using RazorPagesSample.Models
@inject BooksContext _context
@{
   ViewData["Title"] = "Inline";
}
```

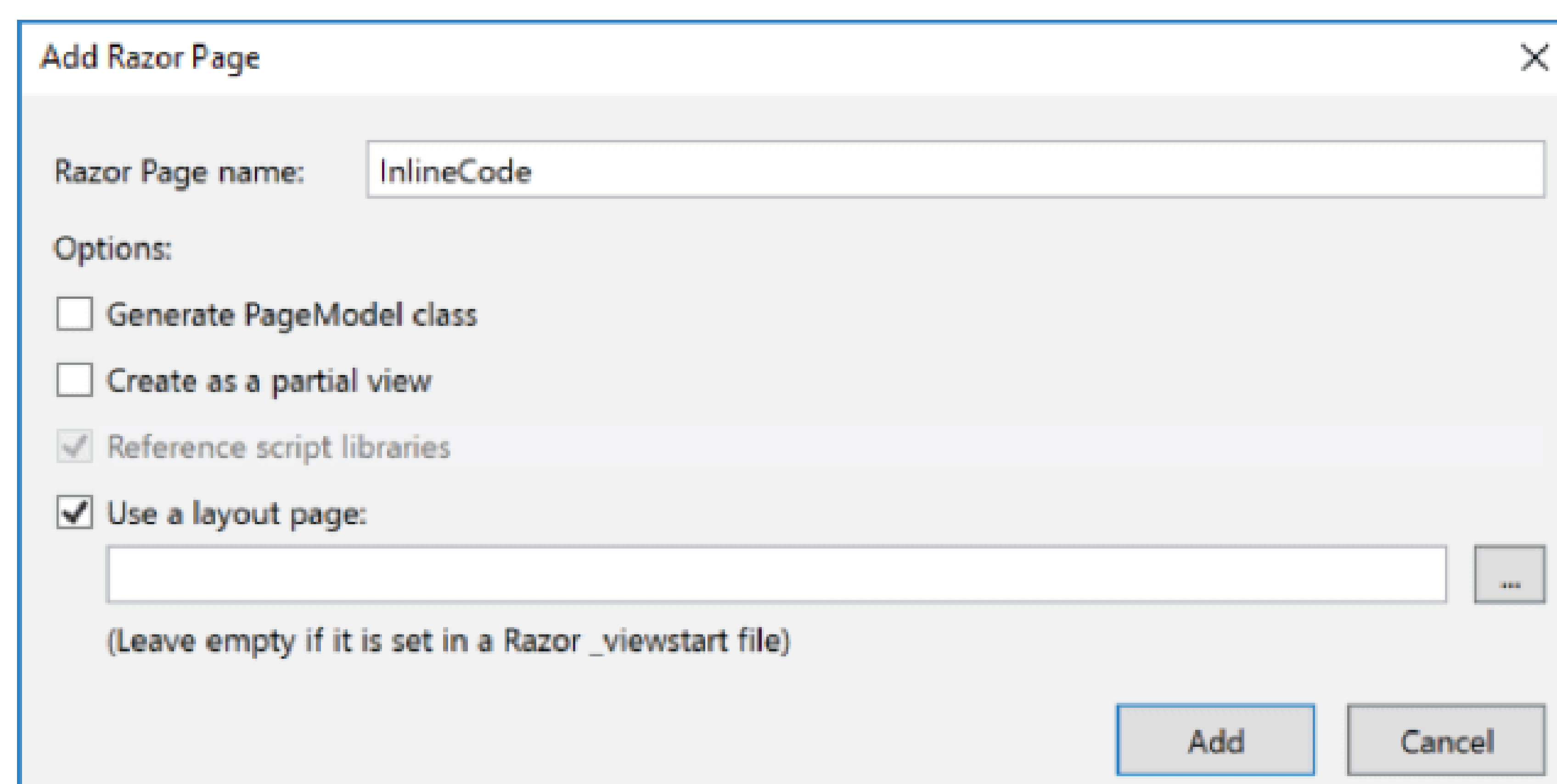



图 31-26

在 Razor 页面中编写内联 C#代码，可以使用@functions 指令。OnGet 方法在页面的每个 GET 请求上调用。在实现代码中，如果数据库还不存在，就创建数据库，并调用 SeedBooks 方法向数据库写入两个 Book 对象。在成功创建数据库之后，将检索图书并将其写入 Books 属性(代码文件 RazorPagesSample/Pages/Inline.cshtml)：

```
@functions
{
  public void OnGet()
  {
    bool created = _context.Database.EnsureCreated();
    if (created) SeedBooks();

    Books = _context.Books.ToList();
  }

  public IEnumerable<Book> Books { get; set; }

  private void SeedBooks()
  {
    _context.Books.Add(new Book
    { Title = "Professional C# 6 and .NET Core 1", Publisher = "Wrox Press" });
    _context.Books.Add(new Book
    { Title = "Professional C# 7 and .NET Core 2", Publisher = "Wrox Press" });
    _context.SaveChanges();
  }
  //...
}
```

**注意：**

在 Razor 页面中，On...方法是根据请求的 HTTP 方法调用的。GET 请求调用 OnGet 方法，POST 请求 OnPost 方法，PUT 请求 OnPut 方法等。还可以向方法名添加后缀，并实现返回 Task 的异步变体。

要显示这些书，请在@function 声明之后的同一页面中，添加 HTML 代码和 Razor 代码，如下所示。使用@foreach，可以迭代并显示 Books 属性中的图书(代码文件 RazorPagesSample/Pages/Inline.cshtml)：

```
@* ... *@

<h2>Inline Razor Page Sample</h2>

@if (Books != null)
{
  <table>
    <thead>
      <tr>
        <th>Title</th>
        <th>Publisher</th>
      </tr>
    </thead>
    <tbody>
      @foreach (var book in Books)
      {
        <tr>
          <td>@book.Title</td>
          <td>@book.Publisher</td>
        </tr>
      }
    </tbody>
  </table>
}
```

下面通过创建表单，发送 POST 请求来扩展这个功能。在同一个页面中，定义一个表单元素，允许用户输入图书标题和出版商。单击 Submit 按钮时，把一个 POST 请求发送到同一个页面(代码文件 RazorPagesSample/Inline.cshtml)：

```
<div>@Message</div>

<form method="post">
  Enter a new book
  <br />
  <input type="text" name="Title" id="Title" />
  <br />
  <input type="text" name="Publisher" id="Publisher" />
  <br />
  <button type="submit">Submit</button>
</form>
```

在@functions 声明中，OnPost 方法定义为响应 POST 请求。在这里，把 Book 属性引用的书写入数据库，并再次检索这些书。Book 属性具有 BindProperty 注释。此属性从 POST 请求 (即表单中输入字段的值) 中获取请求体来创建 Book 对象；因此，这是一种新的模型绑定方式(代码文件 RazorPagesSample/Pages/Inline.cshtml)：

```
@functions
{
  //...

  public void OnPost()
  {
    _context.Books.Add(Book);
    _context.SaveChanges();
    Message = "Book saved";
    Books = _context.Books.ToList();
  }

  [BindProperty()]
  public Book Book { get; set; }

  public string Message { get; set; } = string.Empty;
}
```

默认情况下，使用 BindProperty 特性标记属性不会使属性可用于 GET 请求，这在大多数情况下都是有用的，因为绑定应该只发生在表单的 POST 请求上。但是，还可以将属性 SupportGet 设置为 true，该属性也与 GET 请求绑定。

运行应用程序并在浏览器中输入/Inline 链接，将打开 Razor 页面，创建一个数据库，列出数据库中的图书，并允许用户输入一本新书，如图 31-27 所示——所有这些都定义在一个代码文件中。当然，也可以在同一个 CSHTML 文件中定义 Book 和 BooksContext，这样就可以将所有内容都放在一个文件中。

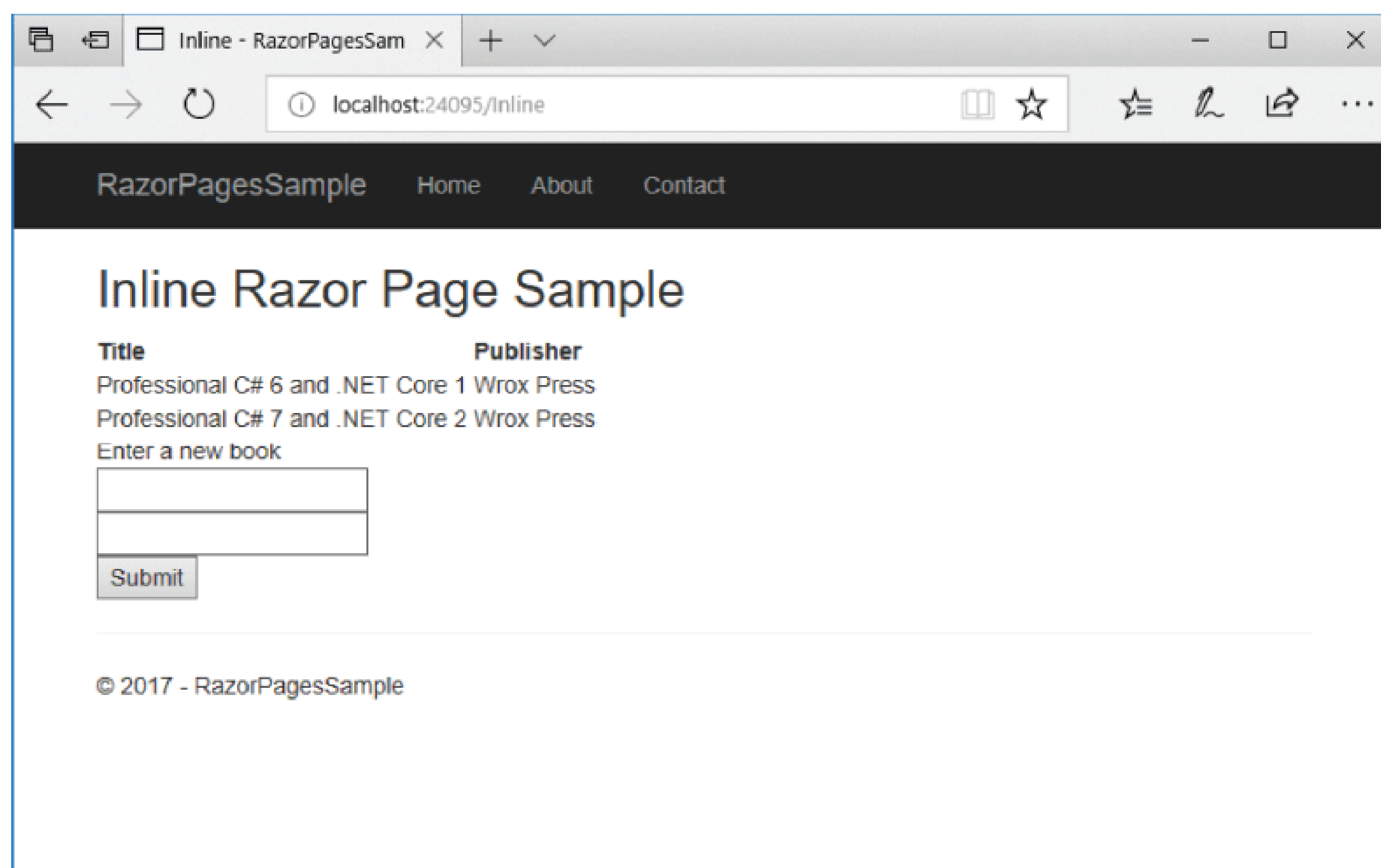



图 31-27

只要没有太多的代码行，把读写图书的所有代码放在一个文件中就是有用的。在.NET Core 中，把所有内容放在一个文件中有益于初学者，但是对于大文件来说，这样会变得很混乱，因为很难找到要更新的行。

## 31.11.4　使用内联代码和页面模型

使用内联代码创建 Razor 页面后的第一个更改是将内联 C#代码更改为一个模型类。现在，在@functions 声明中，定义了派生自类 PageModel 的 InlinePageModel 类。基类 PageModel 在名称空间 Microsoft.AspNetCore.Mvc.RazorPages 中定义。这个类用@model 指令分配给页面——这与把控制器中的信息传递给视图时使用的指令相同。在这个场景中，可以从页面中删除@inject，通过 InlinePageModel 类的构造函数注入 BooksContext。除了构造函数之外，前面示例的@function 中的代码现在在类中定义(代码文件 RazorPagesSample/Pages/InlineWithClasses.cshtml)：

```
@page
@model InlinePageModel
@using Microsoft.AspNetCore.Mvc.RazorPages
@using RazorPagesSample.Models

@{
  ViewData["Title"] = "Inline";
}

@functions
{
  public class InlinePageModel : PageModel
  {
    private readonly BooksContext _context;
    public InlinePageModel(BooksContext context) => _context = context;

    public void OnGet()
    {
      bool created = _context.Database.EnsureCreated();
      if (created) SeedBooks();
```

```
      Books = _context.Books.ToList();
    }

    public IEnumerable<Book> Books { get; set; }

    //...
  }
}
@* ... *@
```

需要用 Razor 代码对 HTML 部分进行小的修改。由于不能直接访问 Books 和 Message 属性，因此需要使用 Model 属性(从@model 指令中创建)(代码文件 RazorPagesSample/Pages/InlineWithClasses.cshtml)：

```
@foreach (var book in Model.Books)
{
  <tr>
    <td>@book.Title</td>
    <td>@book.Publisher</td>
  </tr>
}
```

通过此更改，在 CSHTML 文件中定义 InlinePageModel，C#代码与前一个示例相比有所增长。但是，现在有了一个小步骤，可以将该代码移动到代码隐藏文件中，如下一节所示。

### 31.11.5　使用代码隐藏文件

创建 Razor 页面时，可以选择 Generate PageModel class 选项。这将创建 CodeBehind.cshtml.cs 和代码隐藏文件 CodeBehind.cshtml。CSHTML 文件使用@model 指令引用代码隐藏文件中的类型。这个文件中没有@functions 声明(代码文件 RazorPagesSample/Pages/CodeBehind.cshtml)：

```
@page
@model RazorPagesSample.Pages.CodeBehindModel
@{
  ViewData["Title"] = "Code Behind";
}
<h2>Code Behind Razor Page Sample</h2>
@* ... *@
```

代码隐藏文件包含与内联类相同的代码；不需要除了类名之外的代码更改(代码文件 RazorPagesSample/Pages/CodeBehind.cshtml.cs)：

```
public class CodeBehindModel : PageModel
{
  private readonly BooksContext _context;
  public CodeBehindModel(BooksContext context) => _context = context;

  public void OnGet()
  {
    bool created = _context.Database.EnsureCreated();
    if (created) SeedBooks();

    Books = _context.Books.ToList();
  }
  //...
}
```

C#代码和 HTML 代码分成两个文件。现在就拥有了附近页面的功能，而不是调用多个视图的控制器。

### 31.11.6　页面参数

使用 Razor 页面的简单路由功能，需要一种传递参数的方式。这就是@page 指令所提供的功能：可以在以下代码片段中添加参数作为 int 类型的可选 id 参数(代码文件 RazorPagesSample/Pages/PageWithParameter.cshtml)：

```
@page "{id:int?}"
```

要检索参数，可以修改 OnGet 方法，来接收这个数值(代码文件 RazorPagesSample/Pages/PageWithParameter.cshtml.cs)：

```
public class PageWithParameterModel : PageModel
```

```
{
  public void OnGet(int id = 0)
  {
    Id = id;
  }
  public int Id { get; set; }
}
```

使用 page 指令中定义的路由，可以通过在路由中直接传递值来访问页面：

```
http://localhost:24095/PageWithParameter/42
```

或作为 URL 参数传递：

```
http://localhost:24095/PageWithParameter/?id=42
```

除了更改 OnGet 方法来接收参数之外，也可以将其直接绑定到应用 BindProperty 特性的属性上。请记住设置此特性的 SupportsGet 属性(代码文件 RazorPagesSample /Pages/PageWithParameterAndBinding.cshtml.cs)：

```
public class PageWithParameterAndBindingModel : PageModel
{
  public void OnGet()
  {
  }

  [BindProperty(SupportsGet = true)]
  public int Id { get; set; }
}
```

在学习了 Razor 页面的基础知识之后，就很容易使用 Visual Studio 中的 Razor 页面项模板来创建页面，以便列出、创建、编辑或删除对象，包括使用前面创建的 BooksContext 类的代码隐藏文件。检查文件夹 RazorPagesSample/Pages/Books 内可下载的源代码，查看生成的源文件，其中包括 HTML 辅助程序、Tag Helper 以及访问 EF Core 的代码。

## 31.12 小结

本章介绍了一种使用 ASP.NET Core MVC 框架的最新 Web 技术。这提供了一个健壮的结构，非常适合需要恰当地进行单元测试的大型应用程序。通过本章可以看到，使用 ASP.NET Core MVC 时，提供高级功能十分简单，其逻辑结构和功能的分离使代码很容易理解和维护。

我们还了解了 Razor 页面(这是 ASP.NET Core 2.0 中的新技术)，及其与 ASP.NET Core MVC 的区别。Razor 页面不仅提供了开始使用 ASP.NET Core 的简单方法，它也可能是创建以 HTML 和 JavaScript 为主的 Web 页面所需要的。

在这里，Razor 页面很容易与 ASP.NET Core MVC 混合是很重要的，因为它用于开发 Web API——该 API 用于客户端和服务端之间的通信，而客户端可以是 Web 页面或 WPF、UWP 和 Xamarin 客户端。创建 Web API 是下一章的主题。

# 第 32 章

# Web API

**本章要点**

- ASP.NET Web API 概述
- 创建 Web API 控制器
- 使用存储库和依赖注入
- 调用 REST API 创建.NET 客户端
- 在服务中使用 Entity Framework Core
- 使用 Swagger 创建元数据
- 使用 OData
- 实现 Azure Function

**本章源代码下载地址(wrox.com)：**

打开 www.wrox.com 的 Download Code 选项卡可下载本章源代码。源代码也可以在 API 目录的 https://github.com/ProfessionalCSharp/ProfessionalCSharp7 中找到。本章代码分为以下几个主要的示例文件：

- Book Service Sample
- Book Service Async Sample
- Book Service Client App
- Book Service OData
- Book Service Azure Functions

## 32.1 概述

.NET 3.0 发布 WCF(Windows Communication Foundation)时，WCF 是一种通信技术，替代了.NET 栈中的其他几个技术(其中的两个是.NET Remoting 和 ASP.NET Web 服务)。其目标是只用一种非常灵活的通信技术来满足所有需求。但是，WCF 最初基于 SOAP。现在有许多情形都不需要强大的 SOAP 改进功能。对于返回 JSON 的 HTTP 请求这样的简单情形，WCF 过于复杂。因此在 2012 年引入了另一种技术：ASP.NET Web API。随着 ASP.NET Core 的发布，发布了使用 ASP.NET 技术的 Web API 的第三个重要版本。

ASP.NET MVC 和 ASP.NET Web API 以前有不同的类型和配置(以前的版本是 ASP.NET MVC 5 和 ASP.NET

Web API 2)，但 ASP.NET Core 中的 Web API 与 ASP.NET Core MVC 中的 Web API 相同。

ASP.NET Web API 提供了一种基于 REST(Representational State Transfer)的简单通信技术。REST 是基于一些限制的体系结构样式。下面比较基于 REST 体系结构样式的服务和使用 SOAP 的服务，以了解这些限制。

REST 服务和使用 SOAP 协议的服务都利用了客户端-服务器技术。SOAP 服务可以是有状态的，也可以是无状态的；REST 服务总是无状态的。SOAP 定义了它自己的消息格式，该格式有标题和正文，可以选择服务的方法。而在 REST 中，使用 HTTP 动词 GET、POST、PUT 和 DELETE。GET 用于检索资源，POST 用于添加新资源，PUT 用于更新资源，DELETE 用于删除资源。

本章使用 ASP.NET Core MVC 介绍 Web API 的各个重要方面——创建服务、使用不同的路由方法、创建客户端、使用 OData、保护服务以及使用自定义的宿主。本章还讨论如何通过 ASP.NET Core 使用已创建的相同服务，在 Microsoft Azure Functions 中使用它们，这是使用 C#和.NET 创建 Web API 的另一个选项。

## 32.2 创建服务

首先创建服务。使用.NET Core 时，需要从 ASP.NET Core Web Application 开始，并在如图 32-1 所示的对话框中选择 Web API。这个模板添加了 Web API 需要的文件夹和引用。如果需要 Web 页面和服务，还可以使用模板 Web Application(Model-View-Controller)。使用示例代码，这个项目在解决方案 BooksServiceSample 中命名为 BooksServiceSampleHost。

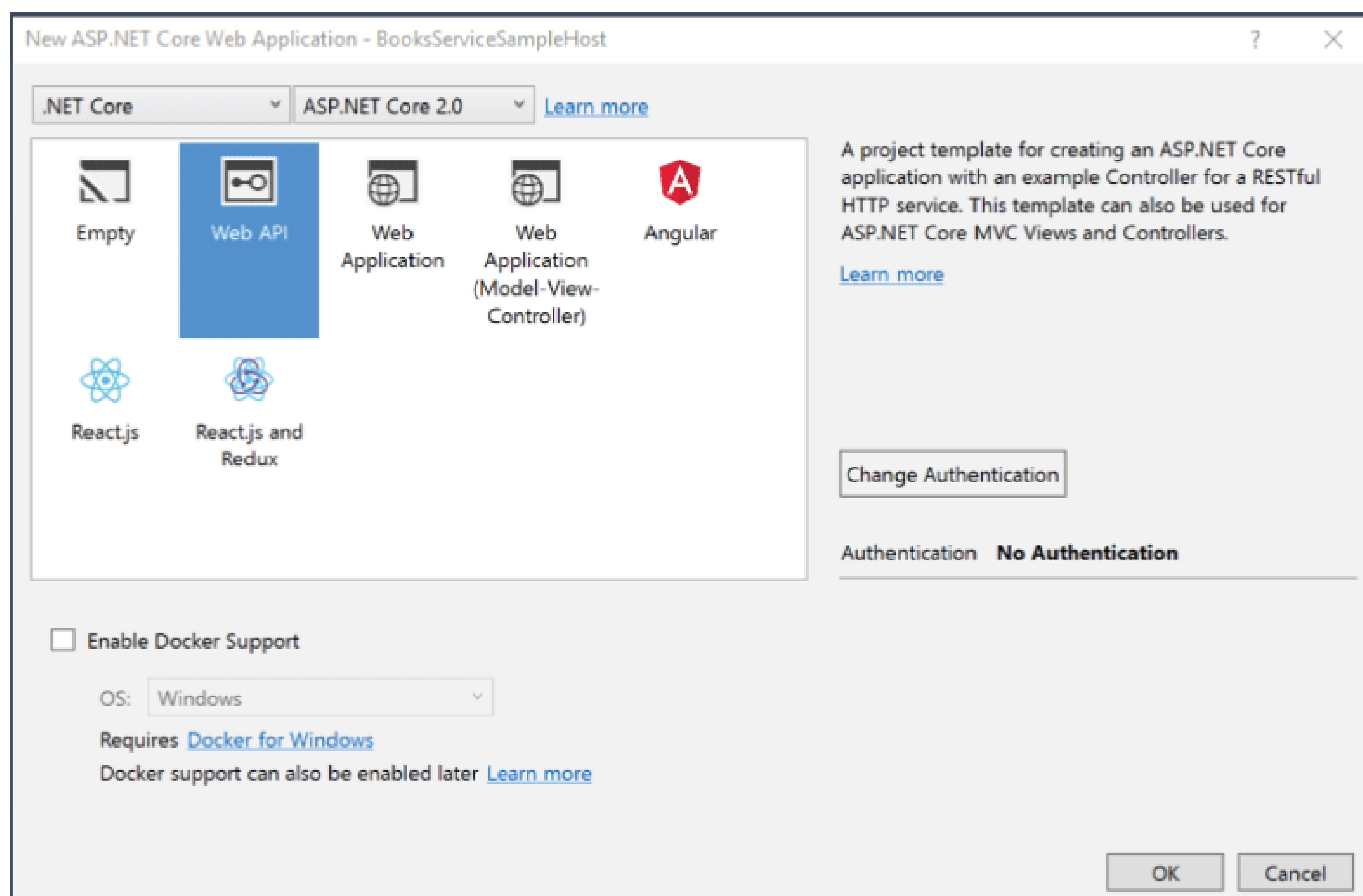



图 32-1

**注意：**

ASP.NET Core MVC 参见第 31 章。ASP.NET Core MVC 的基础核心技术参见第 30 章。

用这个模板创建的目录结构包含创建服务所需要的文件夹。Controllers 目录包含 Web API 控制器。第 31 章介绍过这样的控制器，事实上 Web API 和 ASP.NET Core MVC 使用相同的基础设施。ASP.NET 的.NET Framework 版本不是这样。使用默认模板，ValuesController 是通过一个简单的示例实现创建的。在较大的应用程序中，最好将其分离为多个库。如果创建一个包含服务和模型的库，那么使用来自不同技术(例如，来自 Web API 项目和 Azure Functions)的相同类型是很容易的。Web API(控制器)的实现也可以位于与托管应用程序分离的库中。在 Web 应用程序中，可以使用在应用程序自己的库中实现的控制器。服务(BookServices)和 Web API (APIBookServices)的库都实现为.NET 标准 2.0 库。有了库 APIBookServices，需要添加 NuGet 包

Microsoft.AspNetCore.Mvc.Core 和 Microsoft.AspNetCore.Mvc.ViewFeatures 来实现控制器。在.NET 标准 2.0 库中不可能使用前几章中用过的元数据包 Microsoft.AspNetCore.All，而需要一个.NET Core 2.0 库。

现在，从模型开始。在项目 BooksService 中，Models 目录用于数据模型。可以将实体类型添加到此目录，以及返回模型类型的存储库。

所创建的服务返回图书的章节列表，并允许动态添加和删除章节。

### 32.2.1　定义模型

首先需要一个类型来表示要返回和修改的数据。在 Models 目录中定义的类的名称是 BookChapter，它包含表示一章的简单属性(代码文件 Sync/BookServices/Models/BookChapter.cs)：

```
public class BookChapter
{
  public Guid Id { get; set; }
  public int Number { get; set; }
  public string Title { get; set; }
  public int Pages { get; set; }
}
```

### 32.2.2　创建服务

接下来，创建一个服务。服务提供的方法由接口 IBookChaptersService 定义，用于检索、添加和更新图书章节(代码文件 Sync/BookServices/Services/IBookChaptersService.cs)：

```
public interface IBookChaptersService
{
  void Add(BookChapter bookChapter);
  void AddRange(IEnumerable<BookChapter> chapters);
  IEnumerable<BookChapter> GetAll();
  BookChapter Find(Guid id);
  BookChapter Remove(Guid id);
  void Update(BookChapter bookChapter);
}
```

服务的实现由类 BookChaptersService 定义。书的章节保存在一个集合类中。由于不同客户机请求的多个任务可以并发访问该集合，因此在图书章节中使用类型 ConcurrentDictionary。这个类是线程安全的。Add、Remove 和 Update 方法使用集合来添加、删除和更新图书章节(代码文件 BookServices /Services/BookChaptersService.cs)：

```
public class BookChaptersService : IBookChaptersService
{
  private readonly ConcurrentDictionary<Guid, BookChapter> _chapters =
    new ConcurrentDictionary<Guid, BookChapter>();

  public void Add(BookChapter chapter)
  {
    chapter.Id = Guid.NewGuid();
    _chapters[chapter.Id] = chapter;
  }

  public void AddRange(IEnumerable<BookChapter> chapters)
  {
    foreach (var chapter in chapters)
    {
      chapter.Id = Guid.NewGuid();
      _chapters[chapter.Id] = chapter;
    }
  }

  public BookChapter Find(Guid id)
  {
    _chapters.TryGetValue(id, out BookChapter chapter);
    return chapter;
  }

  public IEnumerable<BookChapter> GetAll() => _chapters.Values;

  public BookChapter Remove(Guid id)
  {
```

```
      BookChapter removed;
      _chapters.TryRemove(id, out removed);
      return removed;
    }

    public void Update(BookChapter chapter) =>
      _chapters[chapter.Id] = chapter;
  }
```

**注意：**

通过示例代码，Remove 方法确保 id 参数传递的 BookChapter 不在字典中。如果字典已经不包含书的章节，那没关系。

如果找不到所传递的图书章节，Remove 方法的另一种实现可以抛出异常。控制器可以更改此错误，以返回 HTTP 未找到的状态码(404)。

Microsoft REST API 指南(https://github.com/microsoft/apiguidelines/blob/master/Guidelines.md)指定 DELETE 请求为幂等性的，因此它应该在多个请求中返回相同的结果。

**注意：**

并发集合详见第 11 章。

因此，第一次访问服务时，可以使用一些示例章节，类 SampleChapters 用章节信息填充图书章节服务(代码文件 Sync/BookServices/Services/SampleChapter.cs)：

```
public class SampleChapters
{
  private readonly IBookChaptersService _bookChaptersService;
  public SampleChapters(IBookChaptersService bookChapterService)
  {
    _bookChaptersService = bookChapterService;
  }

  private string[] sampleTitles = new[]
  {
    ".NET Application Architectures",
    "Core C#",
    "Objects and Types",
    "Object-Oriented Programming with C#",
    "Generics",
    "Operators and Casts",
    "Arrays",
    "Delegates, Lambdas, and Events",
    "Windows Communication Foundation"
  };

  private int[] chapterNumbers = { 1, 2, 3, 4, 5, 6, 7, 8, 44 };

  private int[] numberPages = { 35, 42, 33, 20, 24, 38, 20, 32, 44 };

  public void CreateSampleChapters()
  {
    var chapters = new List<BookChapter>();
    for (int i = 0; i < 8; i++)
    {
      chapters.Add(new BookChapter
      {
        Number = chapterNumbers[i],
        Title = sampleTitles[i],
        Pages = numberPages[i]
      });
    }
    _bookChaptersService.AddRange(chapters);
  }
}
```

在托管应用程序中，引用了库。要使服务可用，需要用依赖注入(DI)容器注册。这是在 Startup 类中完成的。在启动之后，BookChaptersService 和 SampleChapters 服务通过 DI 容器的 AddSingleton 方法注册，为请求

服务的所有客户端创建一个实例。由于 BookChaptersService 注册为单例，因此可以同时从多个线程中访问它；这就是为什么在其实现代码中需要 ConcurrentDictionary 的原因(代码文件 Sync/BookServiceSampleHost/Startup.cs)：

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddMvc();
  //...
  services.AddSingleton<IBookChaptersService, BookChaptersService>();
  services.AddSingleton<SampleChapters>();
}
```

创建示例章节的调用在 Configure 方法中完成。在这里，将注入 SampleChapters 对象，在方法的实现中，将调用 CreateSampleChapters 创建示例章节 (代码文件 Sync/BookServiceSampleHost/Startup.cs)：

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env,
  SampleChapters sampleChapters)
{
  if (env.IsDevelopment())
  {
    app.UseDeveloperExceptionPage();
  }

  app.UseMvc();
  sampleChapters.CreateSampleChapters();
}
```

## 32.2.3　创建控制器

Web API 控制器使用图书章节服务。控制器可以通过 Solution Explorer 上下文菜单 Add New Item | Web API Controller Class 创建。管理图书章节的控制器类被命名为 BookChaptersController。这个类派生自基类 Controller。到控制器的路由使用 Route 特性定义。该路由以 api 开头，其后是控制器的名称，这是没有 Controller 后缀的控制器类名。BooksChapterController 的构造函数需要一个实现 IBookChapterRepository 接口的对象。这个对象是通过依赖注入功能注入的(代码文件 Sync/API BookServices/Controllers/BookChaptersController.cs)：

```
[Produces("application/json", "application/xml")]
[Route("api/[controller]")]
public class BookChaptersController : Controller
{
  private readonly IBookChaptersService _bookChaptersService;
  public BookChaptersController(IBookChaptersService bookChaptersService)
  {
    _bookChaptersService = bookChaptersService;
  }
```

模板中创建的 Get 方法被重命名，并被修改为返回类型为 IEnumerable<BookChapter>的完整集合(代码文件 Sync/APIBookServices/Controllers/BookChaptersController.cs)：

```
// GET api/bookchapters
[HttpGet]
public IEnumerable<BookChapter> GetBookChapters() =>
  _bookChaptersService.GetAll();
```

带一个参数的 Get 方法被重命名为 GetBookChapterById，用 Find 方法过滤存储库的字典。过滤器的参数 id 从 URL 中检索。如果没有找到章节，存储库的 Find 方法就返回 null。在这种情况下，返回 NotFound。NotFound 返回一个 404(未找到)响应。找到对象时，创建一个新的 ObjectResult 并返回它：ObjectResult 返回一个状态码 200，其中包含图书的章节(代码文件 Sync/APIBookServices/Controllers/BookChaptersController.cs)：

```
// GET api/bookchapters/guid
[HttpGet("{id}", Name = nameof(GetBookChapterById))]
public IActionResult GetBookChapterById(Guid id)
{
  BookChapter chapter = _bookChaptersService.Find(id);
  if (chapter == null)
  {
    return NotFound();
  }
  else
  {
    return new ObjectResult(chapter);
```

```
    }
  }
```

**注意：**
**路由的定义参见第 31 章。**

要添加图书的新章节，应添加 PostBookChapter。该方法接收一个 BookChapter 作为 HTTP 体的一部分，反序列化后分配给方法的参数。如果参数 chapter 为 null，就返回一个 BadRequest(HTTP 400 错误)。如果添加 BookChapter，这个方法就返回 CreatedAtRoute。CreatedAtRoute 返回 HTTP 状态码 201(已创建)及序列化的对象。返回的标题信息包含到资源的链接，即到 GetBookChapterById 的链接，其 id 设置为新建对象的标识符(代码文件 Sync/APIBookServices/Controllers/BookChaptersController.cs)：

```
// POST api/bookchapters
[HttpPost]
public IActionResult PostBookChapter([FromBody]BookChapter chapter)
{
  if (chapter == null)
  {
    return BadRequest();
  }
  _bookChaptersService.Add(chapter);
  return CreatedAtRoute(nameof(GetBookChapterById), new { id = chapter.Id },
    chapter);
}
```

更新条目需要基于 HTTP PUT 请求。PutBookChapter 方法在集合中更新已有的条目。如果对象还不在集合中，就返回 NotFound。如果找到了对象，就更新它并返回一个成功的结果状态码 204，其中没有内容(代码文件 Sync/APIBookServices/Controllers/BookChaptersController.cs)：

```
// PUT api/bookchapters/guid
[HttpPut("{id}")]
public IActionResult PutBookChapter(Guid id, [FromBody]BookChapter chapter)
{
  if (chapter == null || id != chapter.Id)
  {
    return BadRequest();
  }
  if (_bookChaptersService.Find(id) == null)
  {
    return NotFound();
  }
  _bookChaptersService.Update(chapter);
  return new NoContentResult();
}
```

对于 HTTP DELETE 请求，从字典中删除图书的章节(代码文件 Sync/APIBookServices/Controllers/BookChaptersController.cs)：

```
// DELETE api/bookchapters/5
[HttpDelete("{id}")]
public void Delete(Guid id) => _bookChaptersService.Remove(id);
```

有了这个控制器，就可以在浏览器上进行第一组测试了。打开链接 http://localhost:1079/api/BookChapters(端口号可能不同)，返回 JSON，如下所示：


```
[{"id":"015b0fb6-1a0f-44ac-ba0d-3d6d743bf4df","number":2,"title":"Objects and Types","pages":33},
{"id":"33cc122a-6be2-48b6-83b1-e913fc10da77","number":0,"title":".NET Application Architectures",
"pages":35},{"id":"47bcfa1e-085e-4d11-9a63-ed2421cad912","number":6,"title":"Arrays","pages":20},
{"id":"069a1755-05da-40d7-96d4-a1422eddfcd1","number":3,"title":"Object-Oriented Programming with
C#","pages":20},{"id":"a11cdc6b-087b-4c69-a4c7-cc48f472c1b0","number":5,"title":"Operators and
Casts","pages":38},{"id":"1b638e90-f553-4635-8b54-b6c5a938ad5d","number":1,"title":"Core
C#","pages":42},{"id":"3871a10e-ca1a-4d63-944e-984d869b9416","number":7,"title":"Delegates, Lambdas,
and Events","pages":32},{"id":"311d1c72-844c-4b0b-a674-6af7b24d7530","number":4,"title":"Generics",
"pages":24}]
```


### 32.2.4　修改响应格式

ASP.NET Web API 的.NET Framework 版本返回 JSON 或 XML，这取决于由客户端请求的格式。在 ASP.NET

Core MVC 中，当返回 ObjectResult 时，默认情况下返回 JSON。如果也需要返回 XML，可以添加一个对 Startup 类的 AddXmlSerializerFormatters 的调用。AddXmlSerializerFormatters 是 IMvcBuilder 接口的一个扩展方法，可以使用流利 API 添加到 AddMvc 方法中(代码文件 Sync/BooksServiceSampleHost/Startup.cs)：

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddMvc().AddXmlSerializerFormatters();
  services.AddSingleton<IBookChaptersService, BookChaptersService>();
  services.AddSingleton<SampleChapters>();
}
```

在控制器中，使用 Produces 特性可以指定允许的内容类型和可选的结果(代码文件 Sync/APIBookServices/Controllers/BookChaptersController.cs)：

```
[Produces("application/json", "application/xml")]
[Route("api/[controller]")]
public class BookChaptersController: Controller
{
  //...
}
```

**注意：**
本章后面的 32.4.2 节“从服务中接收 XML”将介绍如何接收 XML 格式的响应。

### 32.2.5　REST 结果和状态码

表 32-1 总结了服务基于 HTTP 方法返回的结果：

表 32-1

| HTTP 方法 | 说　明 | 请　求　体 | 响　应　体 |
|---|---|---|---|
| GET | 返回资源 | 空 | 资源 |
| POST | 添加资源 | 要添加的资源 | 资源 |
| PUT | 更新资源 | 要更新的资源 | 无 |
| DELETE | 删除资源 | 空 | 空 |

表 32-2 显示了重要的 HTTP 状态码、Controller 方法和返回状态码的实例化对象。要返回任何 HTTP 状态码，可以返回一个 HttpStatusCodeResult 对象，用所需的状态码初始化：

表 32-2

| HTTP 状态码 | Controller 方法 | 类　型 |
|---|---|---|
| 200 OK | Ok | OkResult |
| 201 已创建 | CreatedAtRoute | CreatedAtRouteResult |
| 204 无内容 | NoContent | NoContentResult |
| 400 错误请求 | BadRequest | BadRequestResult |
| 401 未授权 | Unauthorized | UnauthorizedResult |
| 404 未找到 | NotFound | NotFoundResult |
| 任何 HTTP 状态码 | | StatusCodeResult |

所有成功状态码都以 2 开头，错误状态码以 4 开头。状态码列表在 RFC 7231 中可以找到：https://tools.ietf.org/html/rfc7231#section-6.3。

## 32.3 创建异步服务

前面的示例代码使用了一个同步服务。如果使用 Entity Framework Core (EF Core)和存储库，可以使用同步或异步的方法。EF Core 支持两者。然而，许多技术(例如使用 HttpClient 类调用其他服务)只提供了异步的方法。这可能会导致一个异步服务，如 Async 文件夹中的项目 BooksServiceSample 所示。

在异步项目中，IBookChaptersService 已经改为异步的版本。这个接口定义为通过服务访问异步方法，如网络或数据库客户端。所有的方法都返回 Task (代码文件 Async/BooksServiceSample/Services/IBookChaptersService.cs)：

```
public interface IBookChaptersRepository
{
  Task AddAsync(BookChapter chapter);
  Task AddRangeAsync(IEnumerable<BookChapter> chapters);
  Task<BookChapter> RemoveAsync(Guid id);
  Task<IEnumerable<BookChapter>> GetAllAsync();
  Task<BookChapter> FindAsync(Guid id);
  Task UpdateAsync(BookChapter chapter);
}
```

类 BookChaptersService 实现了异步方法。读写字典时，不需要异步功能，所以返回的 Task 使用 FromResult 方法创建(代码文件 Async/BooksServiceSample/Services/BookChaptersService.cs)：

```
public class BookChaptersService: IBookChaptersService
{
  private readonly ConcurrentDictionary<string, BookChapter> _chapters =
    new ConcurrentDictionary<string, BookChapter>();

  public Task AddAsync(BookChapter chapter)
  {
    chapter.Id = Guid.NewGuid();
    _chapters[chapter.Id] = chapter;
    return Task.CompletedTask;
  }

  public Task AddRangeAsync(IEnumerable<BookChapter> chapters)
  {
    foreach (var chapter in chapters)
    {
      chapter.Id = Guid.NewGuid();
      _chapters[chapter.Id] = chapter;
    }
    return Task.CompletedTask;
  }

  public Task<BookChapter> RemoveAsync(Guid id)
  {
    BookChapter removed;
    _chapters.TryRemove(id, out removed);
    return Task.FromResult(removed);
  }

  public Task<IEnumerable<BookChapter>> GetAllAsync() =>
    Task.FromResult<IEnumerable<BookChapter>>(_chapters.Values);

  public Task<BookChapter> FindAsync(Guid id)
  {
    _chapters.TryGetValue(id, out BookChapter chapter);
    return Task.FromResult(chapter);
  }

  public Task UpdateAsync(BookChapter chapter)
  {
    _chapters[chapter.Id] = chapter;
    return Task.CompletedTask;
  }
}
```

API 控制器 BookChaptersController 只需要一些变化，以实现为异步版本。控制器方法也返回一个 Task。这样，就很容易调用存储库的异步方法(代码文件 Async/BooksServiceAsyncSample/Controllers/BookChaptersController.cs)：

```
[Produces("application/json", "application/xml")]
[Route("api/[controller]")]
public class BookChaptersController: Controller
{
  private readonly IBookChaptersService _bookChaptersService;
  public BookChaptersController(IBookChaptersService bookChaptersService)
  {
    _bookChaptersService = bookChaptersService;
  }

  // GET: api/bookchapters
  [HttpGet()]
  public Task<IEnumerable<BookChapter>> GetBookChaptersAsync() =>
    _bookChaptersService.GetAllAsync();

  // GET api/bookchapters/guid
  [HttpGet("{id}", Name = nameof(GetBookChapterByIdAsync))]
  public async Task<IActionResult> GetBookChapterByIdAsync(Guid id)
  {
    BookChapter chapter = await _ bookChaptersService.FindAsync(id);
    if (chapter == null)
    {
      return NotFound();
    }
    else
    {
      return new ObjectResult(chapter);
    }
  }

  // POST api/bookchapters
  [HttpPost]
  public async Task<IActionResult> PostBookChapterAsync(
    [FromBody]BookChapter chapter)
  {
    if (chapter == null)
    {
      return BadRequest();
    }
    await _ bookChaptersService.AddAsync(chapter);
    return CreatedAtRoute(nameof(GetBookChapterByIdAsync),
      new { id = chapter.Id }, chapter);
  }

  // PUT api/bookchapters/guid
  [HttpPut("{id}")]
  public async Task<IActionResult> PutBookChapterAsync(
    Guid id, [FromBody]BookChapter chapter)
  {
    if (chapter == null || id != chapter.Id)
    {
      return BadRequest();
    }
    if (await _ bookChaptersService.FindAsync(id) == null)
    {
      return NotFound();
    }
    await _ bookChaptersService.UpdateAsync(chapter);
    return new NoContentResult();
  }

  // DELETE api/bookchapters/guid
  [HttpDelete("{id}")]
  public async Task DeleteAsync(Guid id) =>
    await _ bookChaptersService.RemoveAsync(id);
}
```

对于客户端来说，控制器实现为同步还是异步并不重要，API 调用是相同的。客户端会为这两种情形创建相同的 HTTP 请求。

## 32.4　创建.NET 客户端

使用浏览器调用服务是处理测试的一种简单方法。客户端常常使用 JavaScript(这是 JSON 的优点)和.NET 客

户端。本书创建一个 Console APP(.NET Core)项目来调用服务。

BookServiceClientApp 的示例代码使用了以下依赖项和名称空间：

**依赖项**

Microsoft.Extensions.DependencyInjection

Microsoft.Extensions.Logging.Console

Newtonsoft.Json

**名称空间**

Microsoft.Extensions.Logging

Newtonsoft.Json

System

System.Collections.Generic

System.Linq

System.Net.Http

System.Net.Http.Headers

System.Runtime.CompilerServices

System.Text

System.Threading.Tasks

System.Xml.Linq

### 32.4.1 发送 GET 请求

要发送 HTTP 请求，应使用 HttpClient 类。在本章中，HttpClient 类用来发送不同的 HTTP 请求。要使用 HttpClient 类，需要添加 NuGet 包 System.Net.Http，打开名称空间 System.Net.Http。要将 JSON 数据转换为.NET 类型，应添加 NuGet 包 Newtonsoft.Json。

> **注意：**
> JSON 序列化和使用 Json.NET 的内容参见网上附加第 2 章。

为了把需要的所有 URL 放在一个地方，UrlService 类为需要的 URL 定义了属性(代码文件 Async/BookServiceClientApp/Services/UrlService.cs)：

```
public class UrlService
{
  public string BaseAddress => "http://localhost:1079/";
  public string BooksApi => "api/BookChapters/";
}
```

> **注意：**
> 需要将 UrlService 类中的 BaseAddress 更改为服务的主机和端口号。当启动服务主机时，可以在浏览器中看到端口号。

在示例项目中，泛型类 HttpClientService 创建为对于不同的数据类型只有一种实现方式。构造函数需要通过 DI 获得 UrlService，使用从 UrlService 中检索的基地址创建 HttpClient (代码文件 Async/BookService ClientApp/Services/HttpClientService.cs)：

```
public abstract class HttpClientService<T> : IDisposable
  where T: class
{
  private HttpClient _httpClient;
  private readonly UrlService _urlService;
  private readonly ILogger<HttpClientService<T>> _logger;

  public HttpClientService(UrlService urlService,
```

```
    ILogger<HttpClientService<T>> logger)
  {
    _urlService = urlService ??
      throw new ArgumentNullException(nameof(urlService));
    _logger = logger ?? throw new ArgumentNullException(nameof(logger));

    _httpClient = new HttpClient();
    _httpClient.BaseAddress = new Uri(_urlService.BaseAddress);
  }
  //...
}
```

**注意：**

*在示例代码中，ILogger 接口用于向控制台写入日志信息。第 29 章详细讨论了日志记录。*

方法 GetInternalAsync 发出一个 GET 请求来接收一组项。该方法调用 HttpClient 的 GetAsync 方法来发送 GET 请求。HttpResponseMessage 包含收到的信息。响应的状态码写入控制台来显示结果。如果服务器返回一个错误，则 GetAsync 方法不抛出异常。异常在方法 EnsureSuccessStatusCode 中抛出，该方法在返回的 HttpResponseMessage 实例上调用。如果 HTTP 状态码是错误类型，该方法就抛出一个异常。响应体包含返回的 JSON 数据。这个 JSON 信息读取为字符串并返回(代码文件 Async/BookServiceClientApp/Services/HttpClientService.cs)：

```
private async Task<string> GetInternalAsync(string requestUri)
{
  if (requestUri == null) throw new ArgumentNullException(nameof(requestUri));
  if (_objectDisposed) throw new ObjectDisposedException(nameof(_httpClient));

  HttpResponseMessage resp = await _httpClient.GetAsync(requestUri);
  LogInformation($"status from GET {resp.StatusCode}");
  resp.EnsureSuccessStatusCode();
  return await resp.Content.ReadAsStringAsync();
}

private void LogInformation(string message,
  [CallerMemberName] string callerName = null) =>
  _logger.LogInformation(
    $"{nameof(HttpClientService<T>)}.{callerName}: {message}");
```

服务器控制器用 GET 请求定义了两个方法：一个方法返回所有章，另一个方法只返回一个章，但是需要章的标识符与 URI。方法 GetAllAsync 调用 GetInternalAsync 方法，把返回的 JSON 信息转换为一个集合，而方法 GetAsync 将结果转换成单个项。这些方法声明为虚拟的，允许在派生类中重写它们(代码文件 Async/BookService-ClientApp/Services/HttpClientService.cs)：

```
public async virtual Task<T> GetAsync(string requestUri)
{
  if (requestUri == null) throw new ArgumentNullException(nameof(requestUri));

  string json = await GetInternalAsync(requestUri);
  return JsonConvert.DeserializeObject<T>(json);
}

public async virtual Task<IEnumerable<T>> GetAllAsync(string requestUri)
{
  if (requestUri == null) throw new ArgumentNullException(nameof(requestUri));

  string json = await GetInternalAsync(requestUri);
  return JsonConvert.DeserializeObject<IEnumerable<T>>(json);
}
```

在客户端代码中不使用泛型类 HttpClientService，而用 BookChapterClientService 类进行专门的处理。这个类派生于 HttpClientService，为泛型参数传递 BookChapter。这个类还重写了基类中的 GetAllAsync 方法，按章号给返回的章排序(代码文件 Async/BookServiceClientApp/Services/BookChapterClientService.cs)：

```
public class BookChapterClientService : HttpClientService<BookChapter>
{
  public BookChapterClientService(UrlService urlService,
    ILogger<BookChapterClientService> logger)
    : base(urlService, logger) { }

  public override async Task<IEnumerable<BookChapter>> GetAllAsync(
```

```
    string requestUri)
  {
    IEnumerable<BookChapter> chapters = await base.GetAllAsync(requestUri);
    return chapters.OrderBy(c => c.Number);
  }
}
```

BookChapter 类包含的属性是用 JSON 内容得到的(代码文件 Async/BookServiceClientApp/Models/BookChapter.cs):

```
public class BookChapter
{
  public Guid Id { get; set; }
  public int Number { get; set; }
  public string Title { get; set; }
  public int Pages { get; set; }
}
```

客户端应用程序的 Main()方法调用不同的方法来显示 GET、POST、PUT 和 DELETE 请求，这些请求使用了 SampleRequest 类中的方法。在此之前，通过调用 ConfigureServices 方法，注册用于 DI 的服务(代码文件 Async/BookServiceClientApp/Program.cs):

```
static async Task Main()
{
  Console.WriteLine("Client app, wait for service");
  Console.ReadLine();
  ConfigureServices();
  var test = ApplicationServices.GetRequiredService<SampleRequest>();

  await test.ReadChaptersAsync();
  await test.ReadChapterAsync();
  await test.ReadNotExistingChapterAsync();
  await test.ReadXmlAsync();
  await test.AddChapterAsync();
  await test.UpdateChapterAsync();
  await test.RemoveChapterAsync();
  Console.ReadLine();
}
```

ConfigureServices()方法在 Microsoft.Extensions.DependencyInjection 容器中注册所需的服务，并配置日志记录，写入控制台(代码文件 Async/BookServiceClientApp/Program.cs):

```
public static void ConfigureServices()
{
  var services = new ServiceCollection();
  services.AddSingleton<UrlService>();
  services.AddSingleton<BookChapterClientService>();
  services.AddTransient<SampleRequest>();
  services.AddLogging(logger =>
  {
    logger.AddConsole();
  });

  ApplicationServices = services.BuildServiceProvider();
}

public static IServiceProvider ApplicationServices { get; private set; }
```

类 SampleRequest 实现了所有的示例方法来调用 BookChapterClientService 的方法。在构造函数中，注入 UrlService 和 BookChapterClientService (代码文件 Async/BookServiceClientApp/SampleRequest.cs):

```
public class SampleRequest
{
  private readonly UrlService _urlService;
  private readonly BookChapterClientService _client;
  public SampleRequest(UrlService urlService,
    BookChapterClientService client)
  {
    _urlService = urlService ??
      throw new ArgumentNullException(nameof(urlService));
    _client = client ?? throw new ArgumentNullException(nameof(client));
  }
  //...
}
```

ReadChaptersAsync()方法从 BookChapterClient 中调用 GetAllAsync()方法来检索所有章，并在控制台显示章

的标题(代码文件 Async/BookServiceClientApp/SampleRequest.cs)：

```
public async Task ReadChaptersAsync()
{
  Console.WriteLine(nameof(ReadChaptersAsync));
  IEnumerable<BookChapter> chapters =
    await _client.GetAllAsync(_urlService.BooksApi);
  foreach (BookChapter chapter in chapters)
  {
    Console.WriteLine(chapter.Title);
  }
  Console.WriteLine();
}
```

运行应用程序(启动服务和客户端应用程序)，ReadChaptersAsync()方法显示了 OK 状态码和章的标题：

```
ReadChaptersAsync
info: BookServiceClientApp.Services.BookChapterClientService[0]
      HttpClientService.GetInternalAsync: status from GET OK
.NET Application Architectures
Core C#
Objects and Types
Object-Oriented Programming with C#
Generics
Operators and Casts
Arrays
Delegates, Lambdas, and Events
```

ReadChapterAsync()方法显示了 GET 请求来检索单章。这样，这一章的标识符就添加到 URI 字符串中(代码文件 Async/BookServiceClientApp/SampleRequest.cs)：

```
public async Task ReadChapterAsync()
{
  Console.WriteLine(nameof(ReadChapterAsync));
  var chapters = await _client.GetAllAsync(_urlService.BooksApi);
  Guid id = chapters.First().Id;
  BookChapter chapter = await _client.GetAsync(Addresses.BooksApi + id);
  Console.WriteLine($" {chapter.Number} {chapter.Title}");
  Console.WriteLine();
}
```

ReadChapterAsync()方法的结果如下所示。它显示了两次 OK 状态，因为第一次是这个方法检索所有的章，之后发送对一章的请求：

```
ReadChapterAsync
info: BookServiceClientApp.Services.BookChapterClientService[0]
      HttpClientService.GetInternalAsync: status from GET OK
Info 0 .NET Application Architectures
: BookServiceClientApp.Services.BookChapterClientService[0]
      HttpClientService.GetInternalAsync: status from GET OK
```

如果用不存在的章标识符发送 GET 请求，该怎么办？具体的处理如 ReadNotExistingChapterAsync()方法所示。调用 GetAsync()方法类似于前面的代码段，但会把不存在的标识符添加到 URI。在 HttpClientHelper 类的实现中，HttpClient 类的 GetAsync()方法不会抛出异常。然而，EnsureSuccessStatusCode 会抛出异常。这个异常用 HttpRequestException 类型的 catch 块捕获。在这里，使用了一个只处理异常码 404(未找到)的异常过滤器(代码文件 Async/BookServiceClientApp/SampleRequest.cs)：

```
private async Task ReadNotExistingChapterAsync()
{
  Console.WriteLine(nameof(ReadNotExistingChapterAsync));
  string requestedIdentifier = Guid.NewGuid().ToString();
  try
  {
    BookChapter chapter = await _client.GetAsync(
      Addresses.BooksApi + requestedIdentifier.ToString());
    Console.WriteLine($" {chapter.Number} {chapter.Title}");
  }
  catch (HttpRequestException ex) when (ex.Message.Contains("404"))
  {
    Console.WriteLine($"book chapter with the identifier " +
      $"{requestedIdentifier} not found");
  }
  Console.WriteLine();
}
```

> **注意：**
> 处理异常和使用异常过滤器的内容参见第 14 章。

方法的结果显示了从服务返回的 NotFound 结果：

```
ReadNotExistingChapterAsync
info: BookServiceClientApp.Services.BookChapterClientService[0]
     HttpClientService.GetInternalAsync: status from GET NotFound
book chapter with the identifier a0f629f4-8c46-4d66-8543-74592dba5d5b not found
```

### 32.4.2 从服务中接收 XML

在 32.2.4 节“修改响应格式”中，XML 格式被添加到服务中。将服务设置为返回 XML 和 JSON，添加 Accept 标题值来接受 application/xml 内容，就可以显式地请求 XML 内容。

具体操作如下面的代码段所示。其中，指定 application/xml 的 MediaTypeWithQualityHeaderValue 被添加到 Accept 标题集合中。然后，结果使用 XElement 类解析为 XML(代码文件 BookServiceClientApp/Services/HttpClientService.cs)：

```
public async Task<XElement> GetAllXmlAsync(string requestUri)
{
  if (requestUri is null) throw new ArgumentNullException(nameof(requestUri));

  using (var client = new HttpClient())
  {
    client.BaseAddress = _baseAddress;
    client.DefaultRequestHeaders.Accept.Add(
      new MediaTypeWithQualityHeaderValue("application/xml"));
    HttpResponseMessage resp = await client.GetAsync(requestUri);
    Console.WriteLine($"status from GET {resp.StatusCode}");
    resp.EnsureSuccessStatusCode();
    string xml = await resp.Content.ReadAsStringAsync();
    XElement chapters = XElement.Parse(xml);
    return chapters;
  }
}
```

> **注意：**
> XElement 类和 XML 序列化参见网上附加第 2 章。

在 SampleRequest 类中，调用 GetAllXmlAsync()方法直接把 XML 结果写到控制台(代码文件 Async/BookService-ClientApp/SampleRequest.cs)：

```
private static async Task ReadXmlAsync()
{
  Console.WriteLine(nameof(ReadXmlAsync));
  XElement chapters = await _client.GetAllXmlAsync(_urlService.BooksApi);
  Console.WriteLine(chapters);
  Console.WriteLine();
}
```

运行这个方法，可以看到现在服务返回了 XML：

```
ReadXmlAsync
info: BookServiceClientApp.Services.BookChapterClientService[0]
     HttpClientService.GetAllXmlAsync: status from GET OK
<ArrayOfBookChapter xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <BookChapter>
    <Id>015b0fb6-1a0f-44ac-ba0d-3d6d743bf4df</Id>
    <Number>2</Number>
    <Title>Objects and Types</Title>
    <Pages>33</Pages>
  </BookChapter>
  <BookChapter>
    <Id>33cc122a-6be2-48b6-83b1-e913fc10da77</Id>
    <Number>0</Number>
    <Title>.NET Application Architectures</Title>
```

```
    <Pages>35</Pages>
  </BookChapter>
  <BookChapter>
    <Id>47bcfa1e-085e-4d11-9a63-ed2421cad912</Id>
    <Number>6</Number>
    <Title>Arrays</Title>
    <Pages>20</Pages>
  </BookChapter>
  <!-... more chapters ...->
</ArrayOfBookChapter>
```

### 32.4.3　发送 POST 请求

下面使用 HTTP POST 请求向服务发送新对象。HTTP POST 请求的工作方式与 GET 请求类似。这个请求会创建一个新的服务器端对象。HttpClient 类的 PostAsync 方法需要用第二个参数添加的对象。使用 Json.NET 的 JsonConvert 类把对象序列化为 JSON。成功返回后，Headers.Location 属性包含一个链接，其中，对象可以再次从服务中检索。响应还包含一个带有返回对象的响应体。在服务中修改对象时，Id 属性在创建对象时在服务代码中填充。反序列化 JSON 代码后，这个新信息由 PostAsync 方法返回(代码文件 Async/BookServiceClientApp/Services/HttpClientService.cs)：

```
public async Task<T> PostAsync(string requestUri, T item)
{
  if (requestUri is null) throw new ArgumentNullException(nameof(requestUri));
  if (item is null) throw new ArgumentNullException(nameof(item));
  if (_objectDisposed) throw new ObjectDisposedException(nameof(_httpClient));

  string json = JsonConvert.SerializeObject(item);
  HttpContent content = new StringContent(json, Encoding.UTF8,
    "application/json");
  HttpResponseMessage resp = await _httpClient.PostAsync(requestUri, content);
  LogInformation($"status from POST {resp.StatusCode}");
  resp.EnsureSuccessStatusCode();
  LogInformation($"added resource at {resp.Headers.Location}");
  json = await resp.Content.ReadAsStringAsync();
  return JsonConvert.DeserializeObject<T>(json);
}
```

在 SampleRequest 类中，可以看到添加到服务的章。调用 BookChapterClient 的 PostAsync()方法后，返回的 Chapter 包含新的标识符(代码文件 Async/BookServiceClientApp/SampleRequest.cs)：

```
private static async Task AddChapterAsync()
{
  Console.WriteLine(nameof(AddChapterAsync));
  var client = new BookChapterClient(Addresses.BaseAddress);
  BookChapter chapter = new BookChapter
  {
    Number = 34,
    Title = "ASP.NET Core Web API",
    Pages = 35
  };
  chapter = await client.PostAsync(Addresses.BooksApi, chapter);
  Console.WriteLine($"added chapter {chapter.Title} with id {chapter.Id}");
  Console.WriteLine();
}
```

AddChapterAsync()方法的结果显示了创建对象的一次成功运行：

```
AddChapterAsync
info added chapter ASP.NET Web API with id b490c5c3-ff30-4ad4-8ca4-7436edfb04c0

: BookServiceClientApp.Services.BookChapterClientService[0]
     HttpClientService.PostAsync: status from POST Created
info: BookServiceClientApp.Services.BookChapterClientService[0]
     HttpClientService.PostAsync: added resource at
       http://localhost:1079/api/BookChapters/b490c5c3-ff30-4ad4-8ca4-7436edfb04c0
```

### 32.4.4　发送 PUT 请求

HTTP PUT 请求用于更新记录，使用 HttpClient 方法 PutAsync()来发送。PutAsync()需要第二个参数中的更新

内容和第一个参数中服务的 URL，其中包括标识符(代码文件 Async/BookServiceClientApp/HttpClientService.cs)：

```
public async Task PutAsync(string requestUri, T item)
{
  if (requestUri is null) throw new ArgumentNullException(nameof(requestUri));
  if (item is null) throw new ArgumentNullException(nameof(item));
  if (_objectDisposed) throw new ObjectDisposedException(nameof(_httpClient));

  string json = JsonConvert.SerializeObject(item);
  HttpContent content = new StringContent(json, Encoding.UTF8,
    "application/json");
  HttpResponseMessage resp = await _httpClient.PutAsync(requestUri, content);
  LogInformation($"status from PUT {resp.StatusCode}");
  resp.EnsureSuccessStatusCode();
}
```

在 SampleRequest 类中，章.NET Application Architectures 更新为另一个标题.NET Applications and Tools (代码文件 Async/BookServiceClientApp/SampleRequest.cs)：

```
public async Task UpdateChapterAsync()
{
  Console.WriteLine(nameof(UpdateChapterAsync));
  var chapters = await _client.GetAllAsync(_urlService.BooksApi);
  var chapter = chapters.SingleOrDefault(
    c => c.Title == ".NET Application Architectures");
  if (chapter != null)
  {
    chapter.Title = ".NET Applications and Tools";
    await _client.PutAsync(_urlService.BooksApi + chapter.Id, chapter);
    Console.WriteLine($"updated chapter {chapter.Title}");
  }
  Console.WriteLine();
}
```

UpdateChapterAsync()方法的控制台输出显示了 HTTP NoContent 结果和更新的章标题：

```
UpdateChapterAsync
info: BookServiceClientApp.Services.BookChapterClientService[0]
      HttpClientService.GetInternalAsync: status from GET OK
info: BookServiceClientApp.Services.BookChapterClientService[0]
      HttpClientService.PutAsync: status from PUT NoContent
updated chapter .NET Applications and Tools
```

### 32.4.5　发送 DELETE 请求

示例客户端的最后一个请求是 HTTP DELETE 请求。调用 HttpClient 类的 GetAsync、PostAsync 和 PutAsync 后，显然发送 DELETE 请求的方法是 DeleteAsync。在下面的代码段中，DeleteAsync()方法只需要一个 URI 参数来识别要删除的对象(代码文件 Async/BookServiceClientApp/Services/HttpClientService.cs)：

```
public async Task DeleteAsync(string requestUri)
{
  if (requestUri is null) throw new ArgumentNullException(nameof(requestUri));
  if (_objectDisposed) throw new ObjectDisposedException(nameof(_httpClient));

  HttpResponseMessage resp = await _httpClient.DeleteAsync(requestUri);
  LogInformation($"status from DELETE {resp.StatusCode}");
  resp.EnsureSuccessStatusCode();
}
```

SampleRequest 类定义了 RemoveChapterAsync()方法(代码文件 Async/BookServiceClientApp/SampleRequest.cs)：

```
public async Task RemoveChapterAsync()
{
  Console.WriteLine(nameof(RemoveChapterAsync));
  var chapters = await _client.GetAllAsync(_urlService.BooksApi);
  var chapter = chapters.SingleOrDefault(
    c => c.Title == "Windows Communication Foundation");
  if (chapter != null)
  {
    await _client.DeleteAsync(_urlService.BooksApi + chapter.Id);
    Console.WriteLine($"removed chapter {chapter.Title}");
  }
  Console.WriteLine();
}
```

运行应用程序时，RemoveChapterAsync()方法首先显示了 HTTP GET 方法的状态，因为是先发出 GET 请求来检索所有的章，然后发出成功的 DELETE 请求来删除 Windows Communication Foundation 章节：

```
RemoveChapterAsync
info: BookServiceClientApp.Services.BookChapterClientService[0]
     HttpClientService.GetInternalAsync: status from GET OK
info: BookServiceClientApp.Services.BookChapterClientService[0]
     HttpClientService.DeleteAsync: status from DELETE OK
removed chapter Windows Communication Foundation
```

# 32.5 写入数据库

第 26 章介绍了如何使用 Entity Framework Core (EF Core)将对象映射到关系上。Web API 控制器可以很容易地使用 DbContext。在示例应用程序中，不需要改变控制器；只需要创建并注册另一个存储库，以使用 EF Core。本节描述所需的所有步骤。

## 32.5.1 使用 EF Core

下面开始访问数据库。为了使用 EF Core 与 SQL Server，需要把 NuGet 包 Microsoft.EntityFrameworkCore.SqlServer 添加到包含服务的库项目中。

前面已经定义了 BookChapter 类。这个类保持不变，用于填充数据库中的实例。映射属性在 BooksContext 类中定义。在这个类中，重写 OnModelCreating 方法，把 BookChapter 类型映射到 Chapters 表，使用数据库中创建的默认唯一标识符定义 Id 列的唯一标识符。Title 列限制为最多 120 个字符(代码文件 Async/BookServiceSample/Models/BooksContext.cs)：

```
public class BooksContext: DbContext
{
  public BooksContext(DbContextOptions<BooksContext> options)
    : base(options) { }

  public DbSet<BookChapter> Chapters { get; set; }

  protected override void OnModelCreating(ModelBuilder modelBuilder)
  {
    modelBuilder.Entity<BookChapter>()
      .ToTable("Chapters")
      .HasKey(c => c.Id);
    modelBuilder.Entity<BookChapter>()
      .Property(c => c.Id)
      .HasColumnType("UniqueIdentifier")
      .HasDefaultValueSql("newid()");
    modelBuilder.Entity<BookChapter>()
      .Property(c => c.Title)
      .HasMaxLength(120);
  }
}
```

对于依赖注入容器，需要添加 EF Core 和 SQL Server 来调用扩展方法 AddEntityFramework 和 AddSqlServer。刚才创建的 BooksContext 也需要注册。使用方法 AddDbContext 添加 BooksContext。在该方法的选项中，传递连接字符串(代码文件 Async/BookServiceSampleHost/Startup.cs)：

```
public async void ConfigureServices(IServiceCollection services)
{
  services.AddMvc().AddXmlSerializerFormatters();
  //...
  services.AddDbContext<BooksContext>(optons =>
    options.UseSqlServer(
      Configuration.GetConnectionString("BooksConnection")));
  //...
}
```

连接字符串本身用托管应用程序项目中的应用程序设置定义(配置文件 Async/BookServiceSampleHost/appsettings.json)：

```
"ConnectionStrings": {
```

```
  "BooksConnection": "server=(localdb)\\mssqllocaldb;database=APIBooksSample;
  trusted_connection=true;MultipleActiveResultSets=true"
}
```

在示例应用程序中，数据库将由 C#代码自动创建。这是通过 Startup 类的 Configure 方法注入 BooksContext，并调用 Database 属性的 EnsureCreated()方法来完成的。如果数据库不存在，则 EnsureCreated()方法创建它。如果创建了数据库，那么将调用 CreateSampleChaptersAsync()方法，向数据库提供示例章节。SampleChapters 是以前内存中章节列表使用的相同服务(代码文件 Async/BooksServiceSampleHost/Startup.cs)：

```
public async void Configure(IApplicationBuilder app, IHostingEnvironment env,
  SampleChapters sampleChapters, BooksContext booksContext)
{
  if (env.IsDevelopment())
  {
    app.UseDeveloperExceptionPage();
  }

  app.UseMvc();
  //...

  bool created = booksContext.Database.EnsureCreated();
  if (created)
  {
    await sampleChapters.CreateSampleChaptersAsync();
  }
}
```

**注意：**

*EF Core 还允许使用 Migration 创建数据库。如何在应用程序中实现 Migration 的信息，请参阅第 26 章。*

### 32.5.2 创建数据访问服务

为了使用 BooksContext，需要创建一个实现接口 IBookChaptersService 的服务。类 DBBookChaptersService 利用 BooksContext，而不是像 BookChaptersService 那样使用内存中的字典(代码文件 Async/BookServiceSample/Models/DBBookChaptersService.cs)：

```
public class DBBookChaptersService, IBookChaptersService
{
  Private readonly BooksContext _booksContext;
  public BookChaptersRepository(BooksContext booksContext)
  {
    _booksContext = booksContext;
  }

  public async Task AddAsync(BookChapter chapter)
  {
    await_booksContext.Chapters.AddAsync(chapter);
    await _booksContext.SaveChangesAsync();
  }

  public Task<BookChapter> FindAsync(Guid id) =>
    _booksContext.Chapters.FindAsyncDefaultAsync(c => c.Id == id);

  public async Task<IEnumerable<BookChapter>> GetAllAsync() =>
    await _booksContext.Chapters.ToListAsync();

  public async Task<BookChapter> RemoveAsync(Guid id)
  {
    BookChapter chapter = await _booksContext.Chapters
      .SingleOrDefaultAsync(c => c.Id == id);
    if (chapter == null) return null;

    _booksContext.Chapters.Remove(chapter);
    await _booksContext.SaveChangesAsync();
    return chapter;
  }

  public async Task UpdateAsync(BookChapter chapter)
  {
    _booksContext.Chapters.Update(chapter);
```

```
    await _booksContext.SaveChangesAsync();
  }
}
```

如果考虑是否要使用上下文，可以阅读第 26 章，它涵盖了 Entity Framework Core 的更多信息。

要使用这个服务，必须在容器的注册表中删除 BookChaptersService(或将其注释掉)，并添加 DBBookChaptersService，使依赖注入容器在要求提供接口 IBookChaptersService 时创建这个类的一个实例(代码文件 Async/BookServiceSampleHost/Startup.cs)：

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddMvc().AddXmlSerializerFormatters();
  // services.AddScoped<IBookChaptersService, BookChaptersService>();
  services.AddScoped<IBookChaptersService, DBBookChaptersService>();
  services.AddScoped<SampleChapters>();

  services.AddDbContext<BooksContext>(options =>
    options.UseSqlServer(
      Configuration.GetConnectionString("BooksConnection")));
  //...
}
```

现在，不改变控制器或客户端，就可以再次运行服务和客户端。根据最初在数据库中输入的数据，可以看到 GET/POST/PUT/DELETE 请求的结果。

## 32.6 用 OpenAPI 或 Swagger 创建元数据

为服务创建元数据允许获得服务的描述，并允许使用这种元数据创建客户端。通过使用 SOAP 的 Web 服务，元数据和 Web 服务描述语言(Web Service Description Language，WSDL)自 SOAP 的早期就已经存在。如今，REST 服务的元数据也在这里。目前它不像 WSDL 那样是一个标准，但描述 API 的最流行的框架是 Swagger(http://www.swagger.io)。自 2016 年 1 月起，Swagger 规范已经更名为 OpenAPI，编写本书时，该标准已经可用于 3.0 版本(http://www.openapis.org)。这个规范可以在 https://github.com/OAI/OpenAPI-Specification/blob/OpenAPI.next/versions/3.0.0.md 上用作 GitHub 存储库。

要给 ASP.NET Web API 服务添加 Swagger 或 OpenAPI，可以使用 Swashbuckle。NuGet 包 Swashbuckle.AspNetCore 是用于 ASP.NET Core 的库。

在添加 NuGet 包之后，需要把 Swagger 添加到 DI 容器中。AddSwaggerGen 是一个扩展方法，可以把 Swagger 服务添加到集合中。为了配置 Swagger，调用方法 SwaggerDoc。传递 Info 对象，可以定义标题、描述、联系人信息等(代码文件 Async/BooksServiceSampleHost/Startup.cs)：

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddMvc().AddXmlSerializerFormatters();
  // services.AddScoped<IBookChaptersService, BookChaptersService>();
  services.AddScoped<IBookChaptersService, DBBookChaptersService>();
  services.AddScoped<SampleChapters>();

  services.AddDbContext<BooksContext>(options =>
    options.UseSqlServer(
      Configuration.GetConnectionString("BooksConnection")));

  services.AddSwaggerGen(options =>
  {
    options.SwaggerDoc("v2", new Info
    {
      Title = "Books Service API",
      Version = "v2",
      Description = "Sample service for Professional C# 7",
      Contact = new Contact { Name = "Christian Nagel",
        Url = "https://csharp.christiannagel.com" },
      License = new License { Name = "MIT License" }
    });
  });
}
```

剩下的就是在 Startup 类的 Configure 方法中配置 Swagger。扩展方法 UseSwagger 添加 Swagger 中间件，指定应该生成一个 JSON 模式文件。

可以用 UseSwagger 配置的默认 URL 是/swagger/{version}/swagger.json。对于前面代码段中配置的文档，URL 是/swagger/v1/swagger.json。方法 UseSwaggerUi 启用了 Swagger 图形用户界面，定义了 URL(代码文件 Async/BooksServiceSampleHost/Startup.cs)：

```
public async void Configure(IApplicationBuilder app, IHostingEnvironment env,
  SampleChapters sampleChapters, BooksContext booksContext)
{
  if (env.IsDevelopment())
  {
    app.UseDeveloperExceptionPage();
  }

  app.UseMvc();
  app.UseSwagger();
  app.UseSwaggerUI(options =>
    options.SwaggerEndpoint("/swagger/v2/swagger.json",
      "Book Chapter Services"));
  //...
}
```

配置 Swagger 后运行应用程序，可以看到服务提供的 API 信息。图 32-2 显示了 BooksServiceSample 提供的 API、Values 服务生成的模板和 BooksService 示例，还可以看到用 Swagger 文档配置的标题和描述。

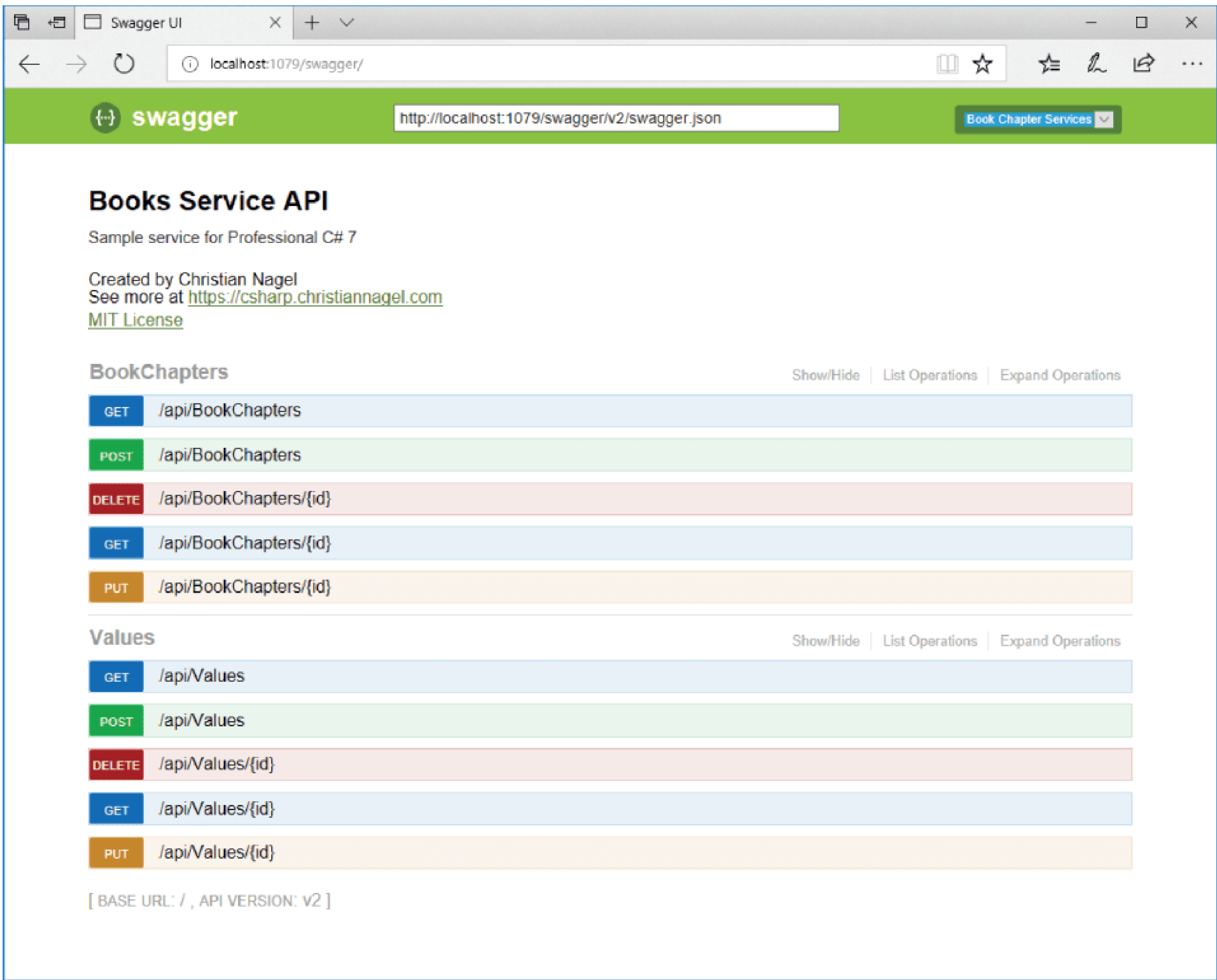



图 32-2

图 32-3 显示了 BookChapters 服务的 GET 请求细节，以及测试 API 调用的 UI。可以看到每个 API 的细节，包括模型。

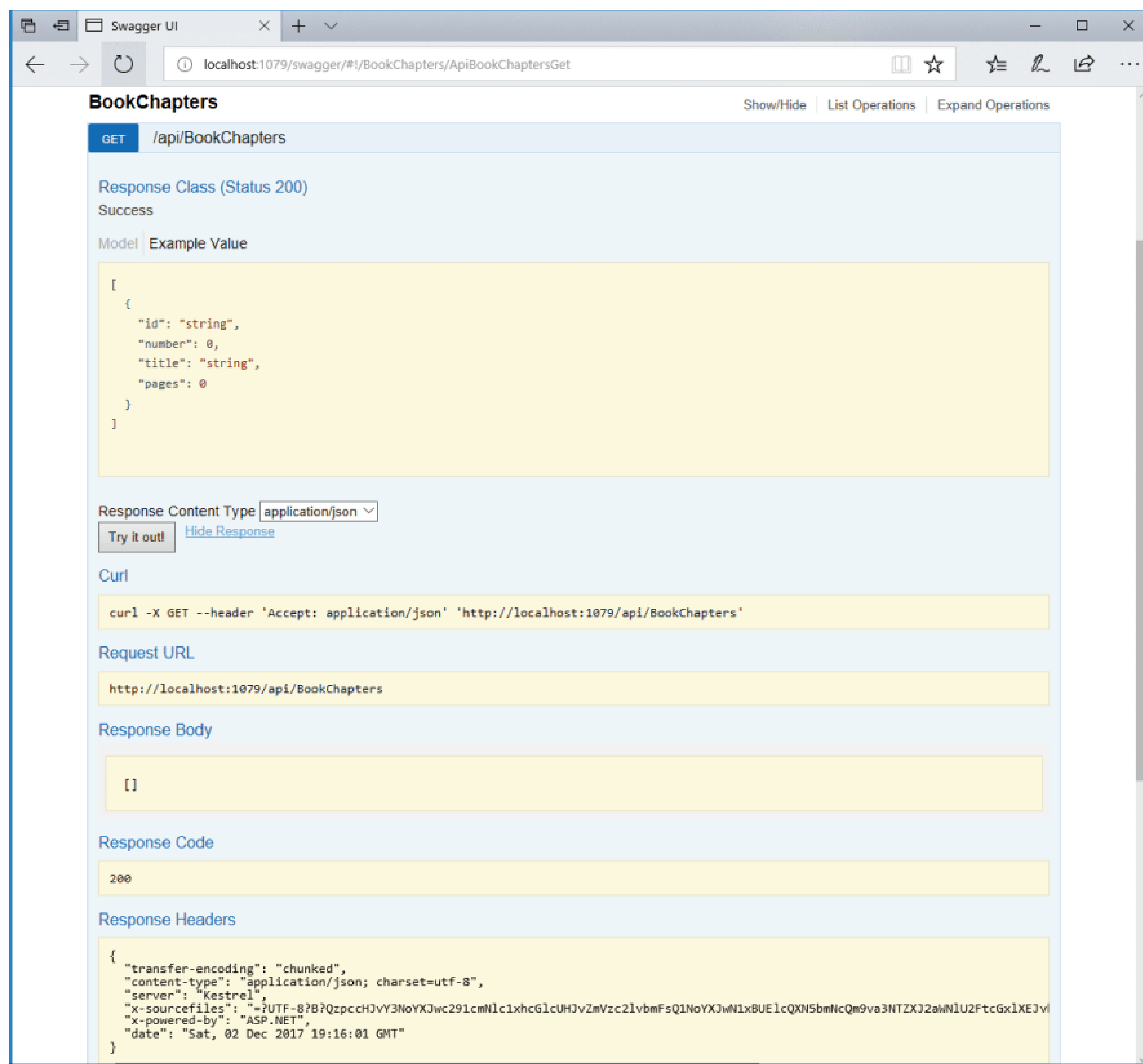



图 32-3

Swagger 允许使用.NET 特性轻松地定制输出。向模型类型添加注释(如 Required 和 DefaultValue)使其成为 JSON 元数据定义，并显示在帮助页面中(代码文件 Async/BooksServiceSample/Models/BookChapter.cs)：

```
public class BookChapter
{
  public Guid Id { get; set; }
  [Required]
  public int Number { get; set; }
  [Required]
  [MaxLength(40)]
  public string Title { get; set; }
  [DefaultValue(0)]
  public int Pages { get; set; }
}
```

通过控制器的操作方法，可以为响应类型指定不同的选项，用 ProducesResponseType 特性在特定的情况下返回模型信息(代码文件 Async/BooksServiceSample/Controllers/BookChaptersController.cs)：

```
[HttpPost]
[ProducesResponseType(typeof(BookChapter), 201)]
[ProducesResponseType(400)]
public async Task<IActionResult> PostBookChapterAsync(
  [FromBody]BookChapter chapter)
{
  //...
}
```

还可以将 XML 注释写入控制器的操作方法中，这将显示在帮助页面中。值得特别关注的是 Swagger 使用

响应元素提供关于方法可能结果的更多信息(代码文件 Async/BooksServiceSample/Controllers/BookChapters-Controller.cs):

```
/// <summary>
/// Creates a BookChapter
/// </summary>
/// <remarks>
/// Sample request:
///     POST api/bookchapters
///     {
///       Number: 42,
///       Title: "Sample Title",
///       Pages: 98
///     }
/// </remarks>
/// <param name="chapter"></param>
/// <returns>A newly created book chapter</returns>
/// <response code="201">Returns the newly created book chapter</response>
/// <response code="400">If the chapter is null</response>
[HttpPost]
[ProducesResponseType(typeof(BookChapter), 201)]
[ProducesResponseType(400)]
public async Task<IActionResult> PostBookChapterAsync(
[FromBody]BookChapter chapter)
{
  //...
}
```

要从 XML 注释生成 XML 文档，需要启用 Project Settings 的 Build 配置，并选择如图 32-4 所示的 XML Documentation File 复选框。此设置指定项目配置文件中的 DocumentationFile 设置(项目文件 Async/BooksServiceSample/BooksServiceSample.csproj):

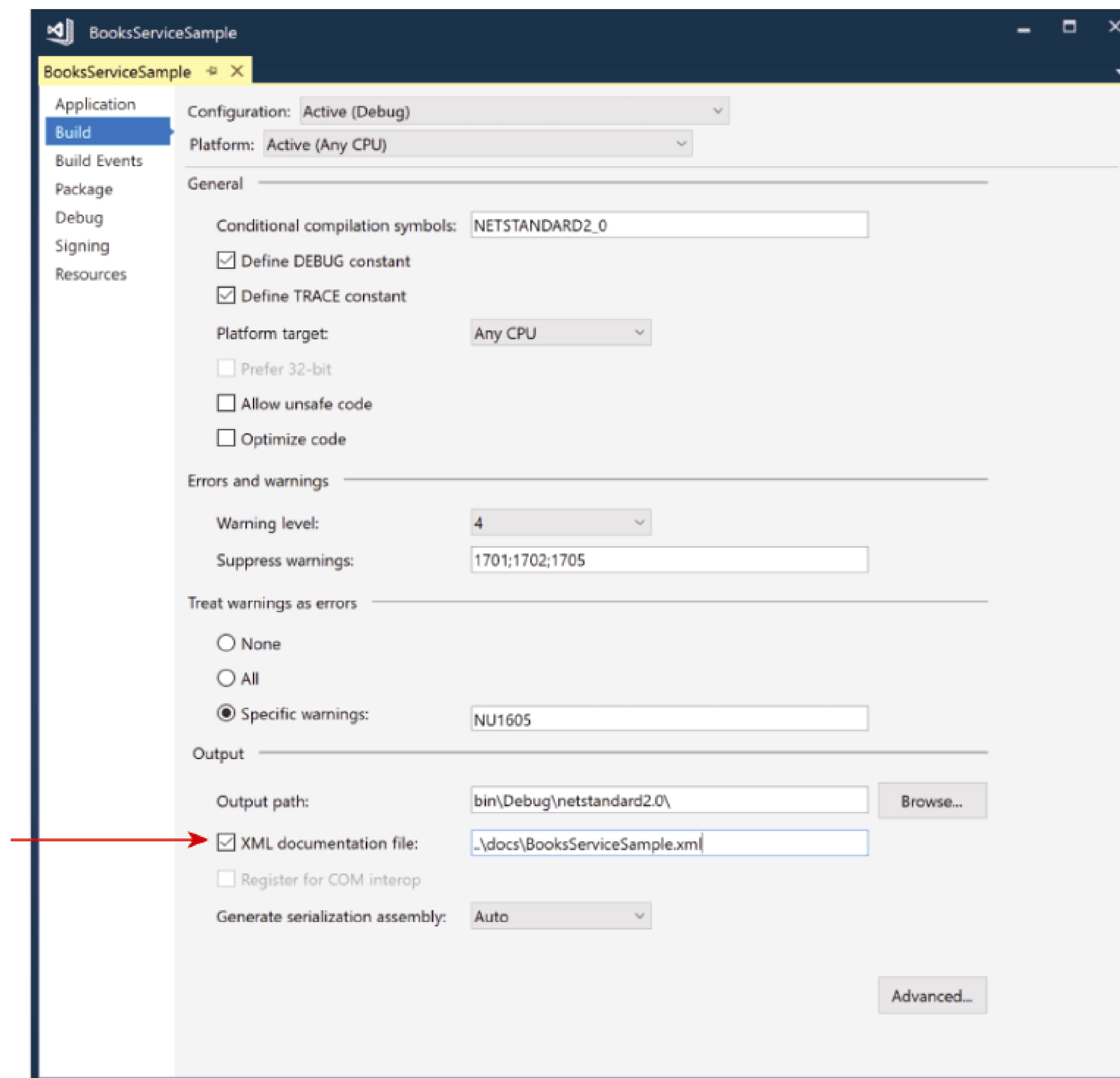



图 32-4

```
<PropertyGroup>
  <DocumentationFile>..\docs\BooksServiceSample.xml</DocumentationFile>
</PropertyGroup>
```

Swagger 现在还需要配置为在调用 AddSwaggerGen 方法时使用这个生成的 XML 文档文件(代码文件 Async/BooksServiceSampleHost/Startup.cs):

```
services.AddSwaggerGen(options =>
{
  options.IncludeXmlComments("../docs/BooksServiceSample.xml");
  options.SwaggerDoc("v2", new Info
  {
    Title = "Books Service API",
    Version = "v2",
    Description = "Sample service for Professional C# 7",
    Contact = new Contact { Name = "Christian Nagel",
      Url = "https://csharp.christiannagel.com" },
    License = new License { Name = "MIT License" }
  });
});
//...
```

通过添加所有这些信息，Swagger 就会显示模型，其中包含注释、来自 XML 文档的信息以及响应代码的特殊信息，如图 32-5 所示。

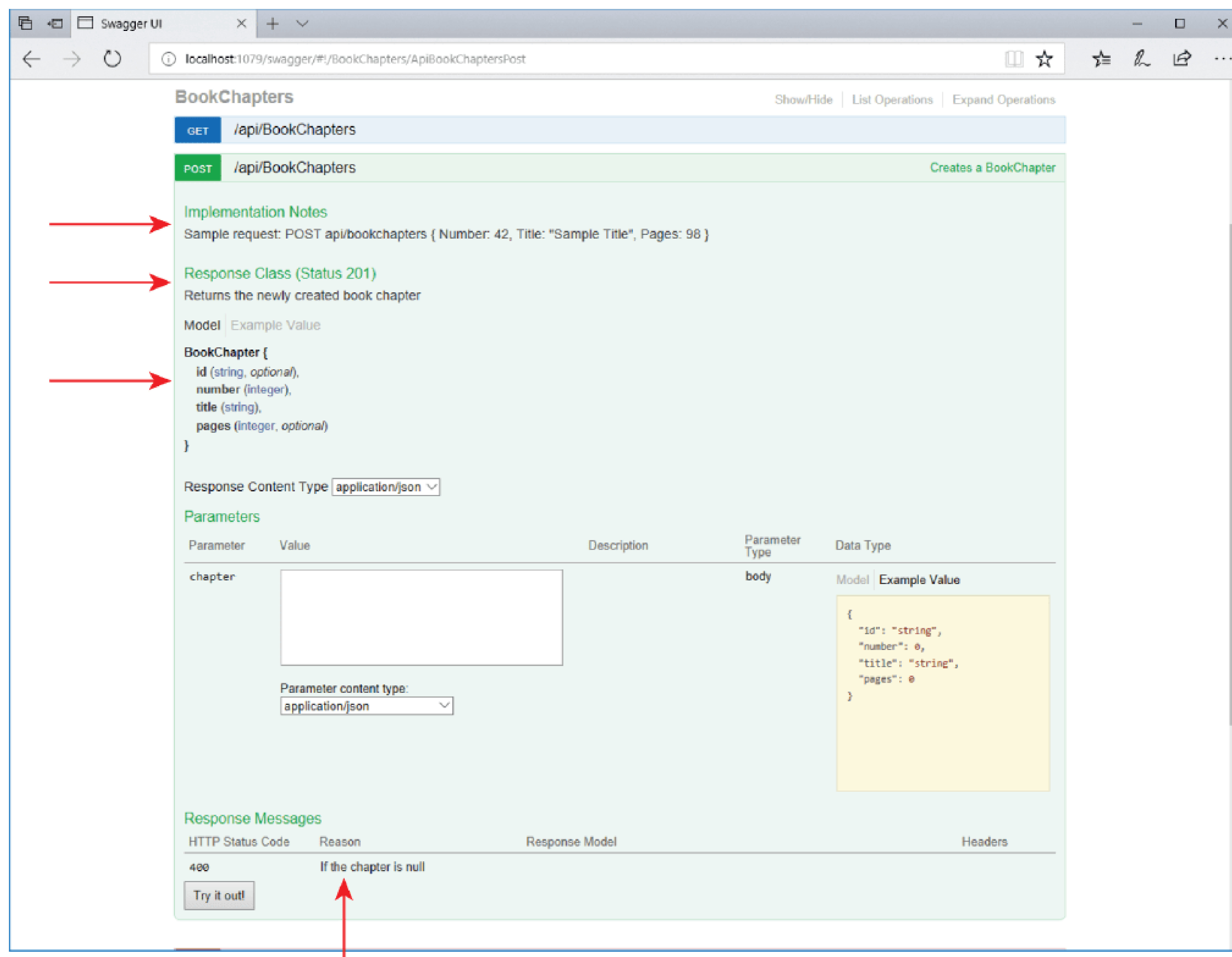



图 32-5

## 32.7　创建和使用 OData 服务

ASP.NET Core 为 OData(Open Data Protocol) 2.1 版本提供了支持。编写本书时，Web API OData 有一个 Beta

版本，请查看本书的 GitHub 页面，获得最近的更新。

OData 通过 HTTP 协议提供了对数据源的 CRUD 访问。发送 GET 请求会检索一组实体数据，POST 请求会创建一个新实体，PUT 请求会更新已有的实体，DELETE 请求会删除实体。前面介绍了映射到控制器中动作方法的 HTTP 方法。OData 基于 JSON 和 AtomPub(一种 XML 格式)进行数据序列化。ASP.NET Core 也直接支持 JSON 和 XML。OData 提供的其他功能有：每个资源都可以用简单的 URL 查询来访问。为了说明其工作方式以及 ASP.NET Web API 如何实现这个功能，下面举例说明，从一个数据库开始。

对于服务应用程序 BooksODataService，为了提供 OData，需要添加 NuGet 包 Microsoft.AspNetCore.OData。示例服务允许查询 Book 和 BookChapter 对象，以及它们之间的关系。

### 32.7.1　创建数据模型

示例服务为模型定义了 Book 和 BookChapter 类。Book 类定义了简单的属性以及与 BookChapter 类型的一对多关系(代码文件 BooksODataService/Models/Book.cs)：

```
public class Book
{
  public Book()
  {
    Chapters = new List<BookChapter>();
  }
  public int Id { get; set; }
  public string Isbn { get; set; }
  public string Title { get; set; }
  public string Publisher { get; set; }
  public List<BookChapter> Chapters { get; set; }
}
```

BookChapter 类定义了简单的属性以及与 Book 类型的多对一关系(代码文件 BooksODataService/Models/BookChapter.cs)：

```
public class BookChapter
{
  public int Id { get; set; }
  public int BookId { get; set; }
  public Book Book { get; set; }
  public string Title { get; set; }
  public int Number { get; set; }
}
```

BooksContext 类定义了 Books 和 Chapters 属性，以及 SQL 数据库关系的定义(代码文件 BooksODataService/Models/BooksContext.cs)：

```
public class BooksContext: DbContext
{
  public DbSet<Book> Books { get; set; }
  public DbSet<BookChapter> Chapters { get; set; }
  protected override void OnModelCreating(ModelBuilder modelBuilder)
  {
    base.OnModelCreating(modelBuilder);
    var bookBuilder = modelBuilder.Entity<Book>();
    bookBuilder.HasMany(b => b.Chapters)
      .WithOne(c => c.Book)
      .HasForeignKey(c => c.BookId);
    bookBuilder.Property(b => b.Title)
      .HasMaxLength(120)
      .IsRequired();
    bookBuilder.Property(b => b.Publisher)
      .HasMaxLength(40)
      .IsRequired(false);
    bookBuilder.Property(b => b.Isbn)
      .HasMaxLength(20)
      .IsRequired(false);
    var chapterBuilder = modelBuilder.Entity<Chapter>();
    chapterBuilder.Property(c => c.Title)
      .HasMaxLength(120);
    chaptersBuilder.HasOne(c => c.Book)
      .WithMany(b => b.Chapters)
```

```
        .HasForeignKey(c => c.BookId);
    }
  }
```

### 32.7.2　创建数据库

要随时使用示例数据创建数据库，使用 CreateBooksService 类注入一个 BooksContext，如果数据库还不存在，则创建它。DatabaseFacade 类上的 EnsureCreated 方法从 BooksContext 的 Database 属性返回，确保数据库存在。如果创建了数据库，则 EnsureCreated 方法返回 true，然后通过调用 CreateSampleBooks 方法给数据库填充一些书籍(代码文件 BooksODataService/Services/CreateBooksService.cs)：

```
public class CreateBooksService
{
  private readonly BooksContext _booksContext;
  public SampleBooks(BooksContext booksContext)
  {
    _booksContext = booksContext;
  }

  public void CreateDatabase()
  {
    bool created = _booksContext.Database.EnsureCreated();
    if (created)
    {
      CreateSampleBooks();
    }
  }
  //...
}
```

图书和章节的示例数据在字段_bookTitles、_bookIsbns 和_chapterTitles 中定义(代码文件 BooksODataService/Services/CreateBooksService.cs)：

```
private string[] _bookTitles = new[]
{
  "Professional C# 7 and .NET Core 2",
  "Professional C# 6 and .NET Core 1.0",
  "Professional C# 5 and .NET 4.5.1"
};

private string[] _bookIsbns = new[]
{
  "978-1-119-44927-0",
  "978-1-119-09660-3",
  "978-1-118-83303-2"
};

private string[][] chapterTitles = new[]
{
  new []
  {
    ".NET Applications and Tools",
    "Core C#",
    "Objects and Types",
    "Object-Oriented Programming with C#",
    "Generics",
    //...
  }
}
```

CreateSampleBooks 方法使用通过私有字段定义的数据，将该信息写入数据库(代码文件 BooksODataService/Services/CreateBooksService.cs)：

```
private void CreateSampleBooks()
{
  for (int i = 0; i < _bookTitles.Length; i++)
  {
    var b = new Book
    {
      Title = _bookTitles[i],
      Isbn = _bookIsbns[i],
      Publisher = "Wrox Press"
```

```
    };
    var chapters = GetChapters(i, b);
    _booksContext.Chapters.AddRange(chapters);
    _booksContext.Books.Add(b);
  }
  int recordsChanged = _booksContext.SaveChanges();
}
```

### 32.7.3 OData 启动代码

在 ASP.NET Core 中，可以轻松地添加 OData 服务。需要将服务添加到依赖注入容器中，并且需要添加中间件，与前面的 ASP.NET Core 章节一样。

在 Startup 类的 ConfigureServices 方法中，CreateBooksService 类注册到 DI 容器中，该容器允许创建数据库。使用 AddDbContext 方法时，EF Core 上下文被注册到 DI 容器中，并配置为使用 SQL Server。对于 OData，调用扩展方法 AddOData。此扩展方法注册 OData 所需的多个服务(代码文件 BooksODataService/Startup.cs)：

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddMvc();
  services.AddTransient<CreateBooksService>();
  services.AddDbContext(options =>
  {
    options.UseSqlServer(Configuration.GetConnectionString("BooksConnection"));
  });
  services.AddOData();
}
```

使用 Configure 方法配置中间件。修改此方法的参数以注入 CreateBooksService。如果数据库还不存在，实现代码中的第一行将创建它。OData 相关代码在 ODataConventionModelBuilder 的创建之后执行。ODataConventionModelBuilder 将.NET 类映射到 Entity Data Model(EDM)。OData 使用 EDM 模型来定义服务公开的数据。OData 路由是通过将 routeBuilder 参数传递给 UseMvc 扩展方法来配置的。通过调用 MapODataServiceRoute 扩展方法指定 OData 路由。该方法的第一个参数指定路由的名称；第二个参数指定路由前缀；第三个参数指定使用先前创建的 ODataConventionModelBuilder 创建模型后返回的 IEdmModel (代码文件 BooksODataService/Startup.cs)：

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env,
  CreateBooksService sampleBooks)
{
  sampleBooks.CreateDatabase();

  if (env.IsDevelopment())
  {
    app.UseDeveloperExceptionPage();
  }

  var builder = new ODataConventionModelBuilder(app.ApplicationServices);
  builder.EntitySet<Book>("Books");
  builder.EntitySet<BookChapter>("BookChapters");

  app.UseMvc(routeBuilder =>
    routeBuilder.MapODataServiceRoute("ODataRoute", "odata",
      builder.GetEdmModel()));
}
```

### 32.7.4 创建 OData 控制器

BooksController 类需要从基类 ODataController 中派生。在下面的代码片段中，Get 方法返回包含 BookChapter 对象的 Book 对象列表。返回 IQueryable 而不是返回 IEnumerable 接口以启用 OData 查询。IEnumerable 接口由 EF Core 中的 DbSet 类实现。从内存数据中返回 List<t>，可以使用 AsQueryable 扩展方法将其转换为 IQueryable。返回一个结果时，如果使用 OData 功能，就需要返回一个 SingleResult 类型的结果。SingleResult.Create()方法创建一个 SingleResult，但是需要 IQueryable 作为参数(代码文件 BooksODataService/Controllers/BooksController.cs)：

```
public class BooksController: ODataController
```

```
{
  private readonly BooksContext _booksContext;
  public BooksController(BooksContext booksContext)
  {
    _booksContext = booksContext;
  }

  public IQueryable<Book> Get() =>
    _booksContext.Books.Include(b => b.Chapters);

  [EnableQuery()]
  public SingleResult<Book> Get([FromODataUri] int id) =>
    SingleResult.Create(_booksContext.Books.Where(b => b.Id == key);
}
```

ChaptersController 的实现与此类似——返回 BookChapter 对象列表和单个 BookChapter(代码文件 BooksODataService/Controllers/ChaptersController.cs):：

```
public class ChaptersController : ODataController
{
  private readonly BooksContext _booksContext;
  public ChaptersController(BooksContext booksContext)
  {
    _booksContext = booksContext;
  }

  public IQueryable<BookChapter> Get() =>
    _booksContext.Chapters.Include(c => c.Book);

  [EnableQuery]
  public SingleResult<BookChapter> Get([FromODataUri] int key) =>
    SingleResult.Create(_booksContext.Chapters.Where(c => c.Id == key));
}
```

除了对 EnableQuery 特性的更改外，不需要对控制器执行其他特殊操作。

### 32.7.5 OData 查询

使用下面的 URL 很容易获得数据库中的所有图书(端口号可能与读者的系统不同)，odata 路由前缀由 Startup 类中的 OData 路由指定，控制器的名称使用约定：

```
http://localhost:50000/odata/Books
```

返回的数据由 JSON 数据和 Book 对象组成：

```
{"@odata.context":"http://localhost:6614/odata/$metadata#Books",
  "value":
    [{"Id":1,"Isbn":"978-1-119-44927-0",
      "Title":"Professional C# 7 and .NET Core 2"},
      "Publisher":"Wrox Press"}
     {"Id":2,"Isbn":"978-1-119-09660-3",
      "Title":"Professional C# 6 and .NET Core 1.0"},
      "Publisher":"Wrox Press"}
     {"Id":3,"Isbn":"978-1-118-83303-2",
      "Title":"Professional C# 5 and .NET 4.5.1"},
      "Publisher":"Wrox Press"}]}
```

同样，使用如下 URL 返回 BookChapter 对象：

```
http://localhost:50000/odata/Chapters
```

要只获取一本书，可以把该书的标识符和 URL 一起传递给方法。这个请求会调用 Get 动作方法，并传递键，返回单一结果：

```
http://localhost:50000/odata/Books(2)
```

可以访问生成的 EDM 信息，这些信息通过 odata 前缀后面的$metadata 字符串传递：

```
http://localhost:50000/odata/$metadata
```

这将使用 ODataConventionModelBuilder 返回从模型中生成的 EDM 信息：

```
<?xml version="1.0" encoding="UTF-8"?>
<edmx:Edmx xmlns:edmx="http://docs.oasis-open.org/odata/ns/edmx" Version="4.0">
  <edmx:DataServices>
```

```
        <Schema xmlns="http://docs.oasis-open.org/odata/ns/edm"
          Namespace="BooksODataService.Models">
          <EntityType Name="Book">
            <Key>
              <PropertyRef Name="Id"/>
            </Key>
            <Property Name="Id" Nullable="false" Type="Edm.Int32"/>
            <Property Name="Isbn" Type="Edm.String"/>
            <Property Name="Title" Type="Edm.String"/>
            <NavigationProperty Name="Chapters"
              Type="Collection(BooksODataService.Models.BookChapter)"/>
          </EntityType>
          <EntityType Name="BookChapter">
            <Key>
              <PropertyRef Name="Id"/>
            </Key>
            <Property Name="Id" Nullable="false" Type="Edm.Int32"/>
            <Property Name="BookId" Type="Edm.Int32"/>
            <Property Name="Title" Type="Edm.String"/>
            <Property Name="Number" Nullable="false" Type="Edm.Int32"/>
            <NavigationProperty Name="Book" Type="BooksODataService.Models.Book">
              <ReferentialConstraint ReferencedProperty="Id" Property="BookId"/>
            </NavigationProperty>
          </EntityType>
        </Schema>
        <Schema xmlns="http://docs.oasis-open.org/odata/ns/edm"
          Namespace="Default">
          <EntityContainer Name="Container">
            <EntitySet Name="Books" EntityType="BooksODataService.Models.Book">
              <NavigationPropertyBinding Target="Chapters" Path="Chapters"/>
            </EntitySet>
            <EntitySet Name="Chapters"
              EntityType="BooksODataService.Models.BookChapter">
              <NavigationPropertyBinding Target="Books" Path="Book"/>
            </EntitySet>
          </EntityContainer>
        </Schema>
      </edmx:DataServices>
    </edmx:Edmx>
```

OData 提供了 ASP.NET Core Web API 支持的强大查询选项。OData 规范允许将参数传递给服务器，进行分页、过滤和排序。下面介绍它们。

每本书都有多个结果。在 URL 查询中，还可以获取书的标题：

```
http://localhost:50000/odata/Books(1)/Title
```

也可以使用关系来检索图书的章节：

```
http://localhost:50000/odata/Books(1)/Chapters
```

要返回数据库中图书章节的数量，可以使用$count 函数：

```
http://localhost:50000/odata/Chapters/$count
```

为了只给客户端返回数量有限的实体，客户端可以使用$top 参数限制数量。也允许使用$skip 进行分页；例如，可以跳过 3 个结果，再提取 3 个结果：

```
http://localhost:50000/odata/Books?$top=3&$skip=3
```

Queryable 特性还有一些命名参数来限制查询，例如最大的 top 和 skip 值、最大的扩展深度以及排序的限制。

为了根据 Book 类型的属性筛选请求，可以将$filter 选项应用于 Book 的属性。为了筛选出 Wrox 出版社出版的图书，可以使用 eq(等于)操作符和$filter 选项：

```
http://localhost:50000/odata/Books?$filter=Publisher eq 'Wrox Press'
```

$filter 选项还可以与 lt(小于)和 gt(大于)操作符一起使用。下面的请求仅返回页数大于 40 的章：

```
http://localhost:50000/odata/Chapters?$filter=Id gt 40
```

为了请求有序的结果，$orderby 选项定义了排序顺序。添加 desc 关键字按降序排序：

```
http://localhost:50000/odata/Book(2)/Chapters?$orderby=Title desc
```

所有这些查询函数都需要显式启用。通过在操作方法上应用 EnableQuery 特性，或者使用 IRouteBuilder 扩展方法来全局地启用 OData 函数和限制。使用下面的代码片段，EnableQuery 特性应用于一个操作方法。在这里，AllowedQueryOptions 属性设置为枚举值 AllowedQueryOptions.All。还可以定义 ODataQueryOptions 类型的参数，以检查通过 URL 请求发送的查询选项，并以编程方式验证：

```
[EnableQuery(AllowedQueryOptions = AllowedQueryOptions.All)]
public IQueryable<Book> Get(ODataQueryOptions options)
{
  ODataValidationSettings settings = new ODataValidationSettings()
  {
    MaxExpansionDepth = 4
  };
  options.Validate(settings);
  var books = _booksContext.Books.Include(b => b.Chapters);
  return books;
}
```

EnableQuery 特性允许指定返回的最大节点数、最大 top 和 skip 值、最大页面大小、是否允许基于哪些属性进行排序，以及允许的逻辑和算术操作符、允许的函数和允许的查询选项。

要使用 Startup 类全局配置选项，可以使用 SetDefaultQuerySettings 方法设置默认值，使用 GetDefaultQuerySettings 方法检索默认值。

## 32.8　使用 Azure Function

使用 ASP.NET Core 创建 Web API 时，可以使用运行 IIS 的 Windows 服务器、运行 Apache 的 Linux 服务器，甚至是没有其他 Web 服务器前端的 Kestrel 服务器来托管它。可以使用 Platform as a Service (PaaS)产品，例如 Azure App Services 来托管 Web API。使用 Azure App Services 时，需要根据 CPU 内核的数量、RAM 的大小和存储大小为服务器实例付费。这些资源是为 Web 应用程序预留的(可以在一个 App Service 实例中运行多个 Web 应用程序)。

根据负载，还有一个托管 Web API 的选项：Consumption 计划或 App Service 计划。对于 App Service 计划，可以在可能已经拥有的 App Service 中运行 Azure Function。另一种变体即 Consumption 计划，也称为 serverless 或 Function as a Service(FaaS)。使用此选项，为运行 Azure Function 所需的请求数和内存付费。根据需要的资源，这个选项可能比使用 App Service 要便宜得多，但也可能更昂贵。还可以在 App Service 实例中运行 Azure Function，该实例将其更改为与 App Service 相同的支付计划。

以无服务器的方式使用 Azure Function 时，它后面仍然有一个服务器。Azure Function 技术总是基于 App Service。但是，以无服务器方式使用它时，不会控制这个服务器，也没有保留 CPU 和内存。这就是价格不同的原因。有关 Microsoft Azure 定价模型的更多信息，请参见 https://azure.microsoft.com/pricing/。

使用 FaaS 托管 Azure Function 有一些限制。Azure Function 最多可以运行 10 分钟。默认超时为 5 分钟，但可以延长到 10 分钟。如果 Azure Function 需要运行更长的时间，就应该在 App Service 计划中托管 Azure Function。

Azure Function 是用静态方法实现的。在多个调用之间共享静态状态。但是，当不需要 Azure Function 时，它就会卸载，当 HTTP 请求再次到达时，它会重新加载和实例化。第一个请求可能需要更长的时间来返回结果。像 App Service 中 always on 这样的选项是不可用于 Consumption 计划的。根据负载，可以使用 Azure Function 自动启动其他机器，这是 Consumption 计划的另一个特性。只需要确保在静态类成员中的调用之间不共享状态。可以使用外部存储特性(如 Azure Storage 或 SQL 数据库)进行状态共享。

### 32.8.1　创建 Azure Function

如果使用通过 DI 使用的服务创建 Web API，并且该服务是在.NET 标准库中定义的，就可以轻松地在 Azure Function 中使用相同的服务。使用 Visual Studio 2017 时，可以在 Add New Project 中选择 Cloud 类别，并选择 Azure Function 模板，来创建 Azure Function 项目。需要安装 Visual Studio 扩展“Azure Functions and Web Jobs

Tools ”，以使用此选项。选择此选项后，可以看到如图 32-6 所示的第一个配置选项。

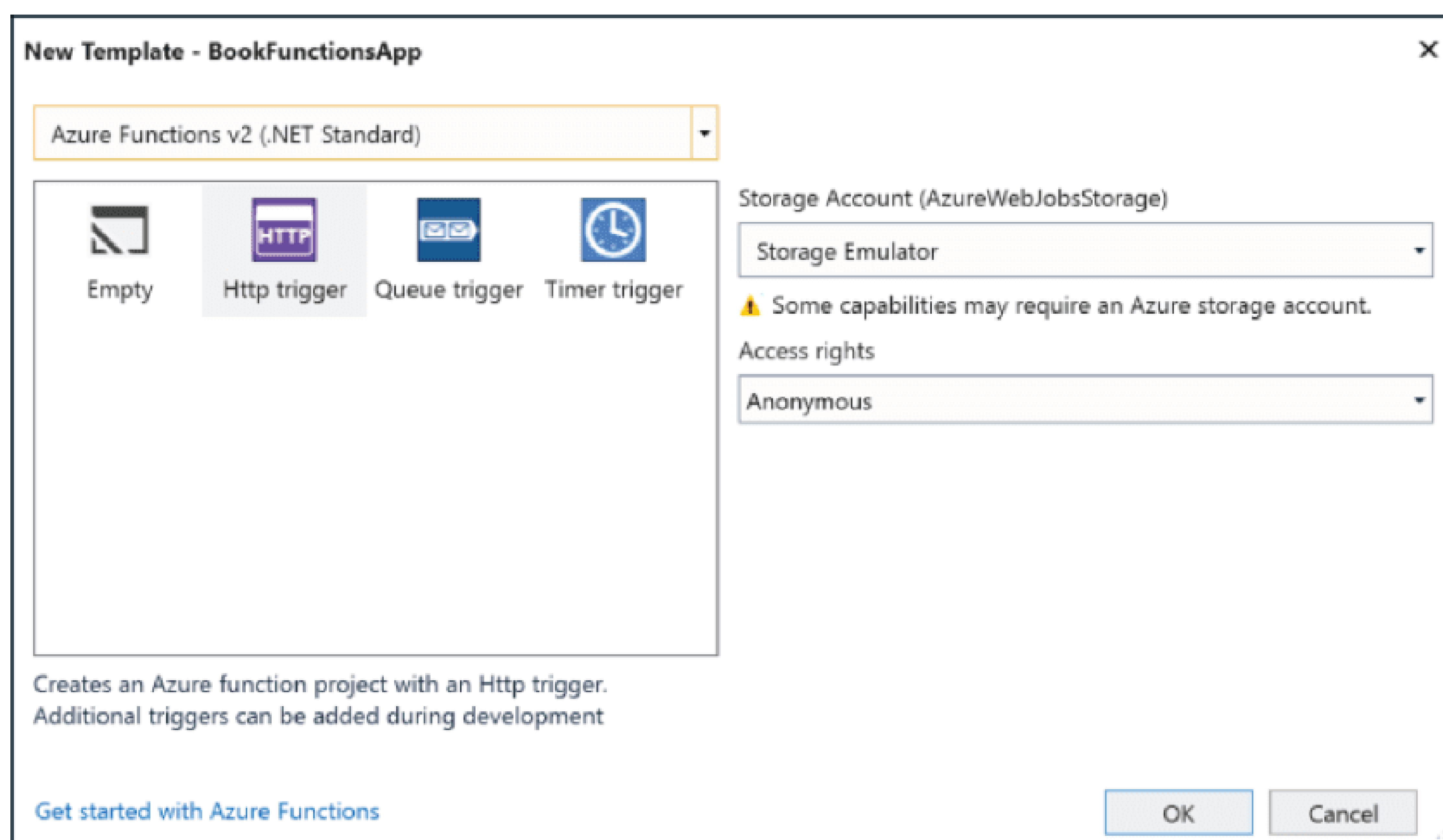



图 32-6

使用这些选项，可以选择在调用函数时触发器的类型。有许多不同的触发器可用。这些触发器的例子包括：把一些数据写入 Azure Cosmos DB、激活一个 WebHook、在 Microsoft Graph 上发生的事件、SMS 到达、到达 Event Hub 的事件、发生在 Blob Storage 中的事件等。最常用的触发器出现在这个对话框中；这些是 HTTP 请求、Azure 存储队列中的项和计时事件。对于存储队列，当消息到达队列时，Function 就可以启动。有了计时器触发器，就可以指定时间间隔，或者在特定的时间启动 Function，比如每个星期六或每个月的第一个星期一。Azure Function 是在间隔时间运行所需后台功能的最好实践——例如，清理或分析数据的存储过程。这一章主要讨论 Web API，这里将使用 HTTP 触发器——触发接收 HTTP 请求的触发器。

要选择的另一个选项是 Azure Function 的版本。在 Azure Functions 1.0 中创建了.NET Framework 库。Azure Functions 2.0 使用.NET Standard 2.0，这通常是最好的选择。只需要注意什么触发器可用于所选的版本。在撰写本文时，Webhook 还不能用于 Azure Functions 2.0，但可以用于 Azure Functions 1.0。

还需要一个带有 Azure Function 的存储账户。要在本地系统上创建和测试 Azure Function，可以使用存储模拟器。在 Azure Function 中写入日志信息需要使用存储账户。

有了访问权限，就指定哪些函数应该可用。可以选择只从其他函数中调用可访问的函数，而不从公共函数调用。这里选择 Anonymous 作为从外部访问 Azure Function 的访问权限。

创建这个项目时，会创建一个引用了 NuGet 包 Microsoft.NET.Sdk.Functions 的.NET Standard 2.0 库。该库包含源文件 Function1.cs，以及 GET 请求的简单 Hello, name 实现。下一节将对其进行更改，以便在 GET、POST 和 PUT 请求上调用 BookChapterService。

## 32.8.2　使用依赖注入容器

虽然 BookChaptersService 很容易通过默认构造函数来实例化，但是对于许多其他服务来说，这是不可能的，比如在构造函数中需要 BooksContext 的 DbBookChaptersService。这就是为什么添加 DI 容器 Microsoft.Extensions DependencyInjection NuGet 包是有用的原因。

BookFunction 类(托管 Azure Function 的类)的静态构造函数调用在 DI 容器中注册服务的 ConfigureServices 方法，使用 SampleChapters 类添加示例章节的 FeedSampleChapters 方法，以及 GetRequiredService 方法，在该方法中，服务将稍后由 Azure Function 的所有特性使用(代码文件 Sync /BookFunctionApp/BookFunction.cs)：

```
public static class BookFunction
{
```

```
    static BookFunction()
    {
      ConfigureServices();
      FeedSampleChapters();
      GetRequiredServices();
    }
    //...
  }
```

ConfigureServices 方法将服务配置到 DI 容器中，这是在使用 ASP.NET Core 时多次看到的功能：

```
private static void ConfigureServices()
{
  var services = new ServiceCollection();
  services.AddSingleton<IBookChaptersService, BookChaptersService>();
  services.AddSingleton<SampleChapters>();
  ApplicationServices = services.BuildServiceProvider();
}

public static IServiceProvider ApplicationServices { get; private set; }
```

要使一些示例章节可用，但不需要创建数据库，CreateSampleChapters 方法使用 BookChaptersService 创建一些内存中的章节。在生产中使用 Azure Function 时，请记住不要在内存中共享状态。而这里使用它，是因为这样更容易演示这个例子。在很短的时间内，这些数据一直存在，但是当函数的空闲时间足够长，或者由于同时创建多个实例有较高的负载时，就可能会得到意想不到的结果。要使用数据库获得稳定的结果，只需要将服务注册从 BookChaptersService 更改为 DbBookChaptersService，并添加 EF Core 上下文：

```
private static void FeedSampleChapters()
{
  var sampleChapters =
    ApplicationServices.GetRequiredService<SampleChapters>();
  sampleChapters.CreateSampleChapters();
}
```

从 GET、POST 和 PUT 请求到 Azure Function，都需要 IBookChaptersService；这就是为什么要在静态变量中检索和存储此服务的原因。使用静态构造函数调用 GetRequiredServices 时，每次重新启动主机时，都会调用此方法：

```
private static void GetRequiredServices()
{
  s_bookChaptersService =
    ApplicationServices.GetRequiredService<IBookChaptersService>();
}

private static IBookChaptersService s_bookChaptersService;
```

在完成 Azure Function 的设置之后，可以在下一节中实现主要功能。

### 32.8.3　实现 GET、POST 和 PUT 请求

Azure Function 的核心是用静态 Run 方法定义的。函数的名称由 FunctionName 特性定义。参数通过触发器的类型来区分。示例代码使用 HttpTrigger 特性指定 HTTP 请求上的触发器。由于这个特性，Run 方法的第一个参数类型是 HttpRequest。此类型包含 HTTP 请求的信息，并允许发送 HTTP 响应。HttpTrigger 特性指定在创建应用程序时指定的 AuthorizationLevel，并在 Azure Function 应该被激活时，在其后面添加一个 HTTP 谓词的可变参数列表。还可以使用参数指定此 Azure Function 的路由信息。使用路由定义的参数也可以作为参数添加到 Run 方法中。Run 方法的最后一个参数是 TraceWriter。此写入器用于将信息记录到创建应用程序时指定的 Azure 存储账户中。Run 方法实现后，根据接收到的 HTTP 方法调用 DoGet、DoPost 和 DoPut 方法(代码文件 Sync/BookFunctionApp/BookFunction.cs)：

```
[FunctionName("BookFunction")]
public static IActionResult Run([HttpTrigger(AuthorizationLevel.Anonymous,
  "get", "post", "put", Route = "null")]HttpRequest req, TraceWriter log)
{
```

```
    log.Info("C# HTTP trigger function processed a request.");

    IActionResult result = null;
    switch (req.Method)
    {
      case "GET":
        result = DoGet(req);
        break;
      case "POST":
        result = DoPost(req);
        break;
      case "PUT":
        result = DoPut(req);
        break;
      default:
        result = new BadRequestResult();
        break;
    }
    return result;
  }
```

通过 GET 请求，客户端可以检索所有图书章节，或者只检索一个章节。如果 HTTP URL 包含带有 Id (/?Id=Guid)的查询，则解析标识符，并调用 IBookChaptersService 的 Find 方法。根据 Find 方法的结果，要么返回 NotFoundResult，要么返回包含 HTTP 主体中图书章节的 OkObjectResult。如果 Id 不是查询的一部分，则使用 GetAll 方法检索所有图书章节，并将结果列表放入 OkObjectResult 的构造函数中(代码文件 Sync/BookFunctionApp/BookFunction.cs)：

```
  private static IActionResult DoGet(HttpRequest req)
  {
    string id = req.Query["Id"];
    if (id != null)
    {
      Guid guid = Guid.Parse(id);
      var chapter = s_bookChaptersService.Find(guid);
      if (chapter == null)
      {
        return new NotFoundResult();
      }
      return new OkObjectResult(chapter);
    }
    else
    {
      var chapters = s_bookChaptersService.GetAll();
      return new OkObjectResult(chapters);
    }
  }
```

使用 HTTP POST 请求，调用 DoPost 方法。POST 请求包括请求的 HTTP 主体中的新书章节。可以通过访问 HttpRequest 的 Body 属性来检索 HTTP 主体。Body 属性的类型是 Stream，它可以放在 StreamReader 类的构造函数中。使用 StreamReader，通过调用 ReadToEnd 检索完整的 JSON 字符串。接着在 Newtonsoft.Json 的帮助下，将这个 JSON 字符串转换为 BookChapter。然后将转换后的 BookChapter 传递给 IBookChaptersService 的 Add 方法(代码文件 Sync/BookFunctionApp/BookFunction.cs)：

```
  private static IActionResult DoPost(HttpRequest req)
  {
    string json = new StreamReader(req.Body).ReadToEnd();
    BookChapter chapter = JsonConvert.DeserializeObject<BookChapter>(json);
    s_bookChaptersService.Add(chapter);
    return new OkResult();
  }
```

**注意：**

流可参阅第 22 章。

更新 BookChapter 对象的 HTTP PUT 请求与以前的 HTTP POST 请求非常相似。这一次只是调用 IBookChaptersService 的 Update 方法(代码文件 Sync/BookFunctionApp/BookFunction.cs)：

```
private static IActionResult DoPut(HttpRequest req)
{
  string json = new StreamReader(req.Body).ReadToEnd();
  BookChapter chapter = JsonConvert.DeserializeObject<BookChapter>(json);
  s_bookChaptersService.Update(chapter);
  return new OkResult();
}
```

有了这些，就可以使用通过 ASP.NET Core 提供服务时已经实现的所有功能。使用服务的一个小部分，服务不需要任何更改。接下来，运行并发布 Azure Function。

## 32.8.4　运行 Azure Function

在 Visual Studio 中运行应用程序时，一个控制台窗口显示了 Azure Function 的徽标(参见图 32-7)，并显示了使用 URL 访问 HTTP 服务的侦听器的输出。现在可以使用浏览器发出 GET 请求，并测试 Azure Function。对于测试 POST 和 PUT 请求，可以调整先前创建的客户端来调用 Azure Function，也可以使用 Postman 之类的工具(https://www.getpostman.com)创建 POST 和 PUT 请求。这也是创建集成和运行集成测试的好工具。

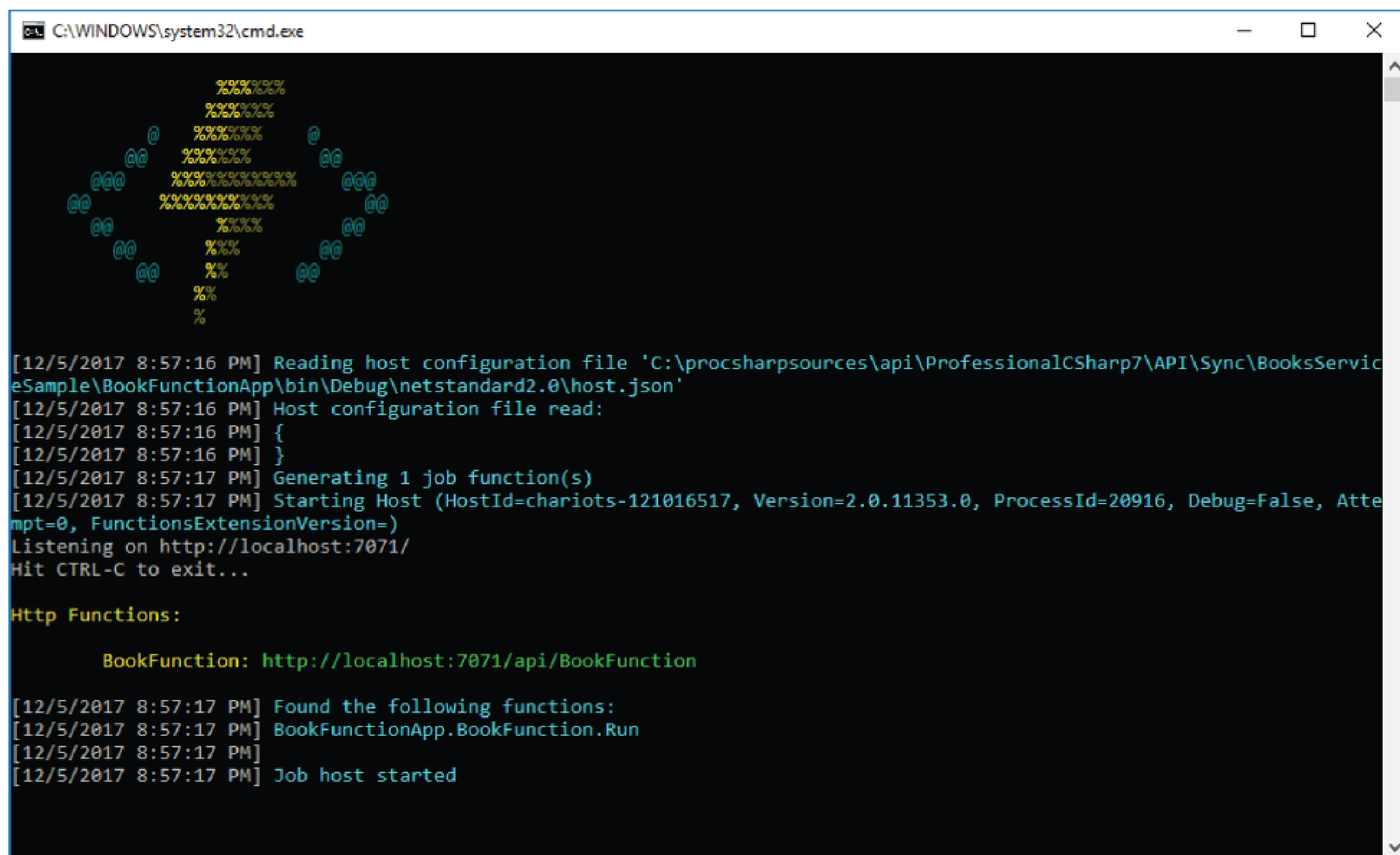



图 32-7

运行应用程序时，会发现 bin/Debug/netstandard2.0/BookFunction 目录下的文件 function.json。此文件描述了在 Microsoft Azure 上发布时部署的 Azure Function。使用.NET 标准库时，function.json 的信息来自使用 Run 方法指定的注释；可以看出，它列出了触发器的类型、触发器的配置，如 HTTP 方法和 Azure Function 的入口点：

```
{
  "generatedBy": "Microsoft.NET.Sdk.Functions.Generator-1.0.6",
  "configurationSource": "attributes",
  "bindings": [
    {
      "type": "httpTrigger",
      "methods": [
        "GET",
        "POST",
        "PUT"
```

```
        ],
        "authLevel": "anonymous",
        "name": "req"
      }
    ],
    "disabled": false,
    "scriptFile": "../bin/BookFunctionApp.dll",
    "entryPoint": "BookFunctionApp.BookFunction.Run"
  }
```

成功地在本地运行应用程序后，就可以将其发布到 Microsoft Azure，或者发布到 Consumption 或 App Service 计划中。

## 32.9　小结

本章使用 ASP.NET Core 描述了 Web API 的功能。这种技术允许使用 HttpClient 类创建服务，并在任何客户端(无论是 JavaScript 还是.NET 客户端)调用。返回 JSON 或 XML，但目前 JSON 是首选格式。

依赖注入已经用于本书的几章，尤其是第 20 章。本章介绍了很容易把使用字典的、基于内存的存储库替换为使用 EF Core 的存储库。

本章还介绍了 OData，它使用资源标识符，很容易引用树中的数据。

除了使用 ASP.NET Core 托管 Web API 之外，还使用了 Azure Function 创建与前面相同的服务。服务是独立于托管技术实现的，因此很容易创建一个小的 facade 并托管来自 Azure Function 的服务，在 Microsoft Azure 上托管 Web API 时，Azure Function 提供了不同的成本模型。

下一章开始本书的第 IV 部分，也是介绍如何使用 XAML 创建 Windows 应用程序的第一章。

# 第 IV 部分

# 应 用 程 序

# 第 33 章

# Windows 应用程序

**本章要点**

- XAML 概论
- 控件
- 已编译的数据绑定
- 导航
- 布局面板

**本章源代码下载：**

打开 www.wrox.com 的 Download Code 选项卡可下载本章源代码。源代码也可以在 Windows 目录的 https://github.com/ProfessionalCSharp/ProfessionalCSharp7 中找到。

本章代码分为以下几个主要的示例文件：

- IntroXAML
- ControlsSample
- ParallaxViewSample
- PageLayouts
- NavigationSample
- LayoutPanels

## 33.1 Windows 应用程序简介

Windows 应用程序使用.NET Core，但是 Web 应用程序和 ASP.NET Core 有很大的区别。Windows 应用程序只在 Windows 平台上运行，在 Windows 10 上运行。这些应用程序不仅适用于桌面，也适用于 Xbox、HoloLens 和 Raspberry PI。

**注意：**

*要在 iPhone 和 Android 上使用 XAML 创建应用程序，请阅读第 37 章。但是，一定要先读本章，因为第 37 章只解释了差异。*

### 33.1.1 Windows 运行库

Windows 应用程序还利用了 Windows 运行库(Windows Runtime，WinRT)。Windows 运行库是使用 C++和新一代 COM 对象创建的平台。因此，Windows 运行库不仅适用于.NET 应用程序，也适用于 C++和使用 JavaScript 创建的应用程序。为了从这些不同的平台上访问 Windows 运行库，还创建了一个兼容层：语言投影。通过语言投影，Windows 运行库提供的 API 看起来像.NET API。

运行库看起来像.NET 是由于语言投影。Windows 运行库的元数据以与.NET 相同的形式创建。可以使用相同的工具(例如 ildasm)在 Windows 运行库中读取元数据信息。使用 COM 动态读取元数据已经有不短的历史了。那时，可以在类型库中访问元数据。这种元数据技术不如用.NET 实现时的元数据强大。在.NET 中，元数据可以使用自定义属性进行扩展，并且可以使用反射进行访问 (请参阅第 16 章)。Windows 运行库现在使用与.NET 相同的元数据格式。因此，可以使用 ildasm 命令行打开.winmd 文件(Windows 运行库的元数据文件)，查看带有参数的 API 调用。Windows 元数据文件可以在目录%ProgramFiles(x86)\Windows Kits\10\References\中找到。

语言投影将 Windows 运行库类型映射到.NET 类型上。例如，在文件 Windows.Foundation.FoundationContract.winmd 中，名称空间 Windows.Foundation.Collections 包含 IIterable 和 IIterator 接口。这些接口看起来非常类似于.NET 接口 IEnumerable 和 IEnumerator。实际上，它们是用语言投影自动映射的。

Windows 运行库不包含任何集合。相反，集合是由使用 Windows 运行库的不同平台实现的，例如 C++、JavaScript 和.NET。

并不是所有的协定接口都可以直接映射。第 22 章展示了名称空间 Windows.Storage.Streams 中带有 Windows 运行库的文件和流。要将 Windows 流与.NET 流一起使用，可以使用扩展方法，如 AsStream、AsStreamForRead 和 AsStreamForWrite。

**注意：**
使用.NET Core API 和 Windows Runtime API 创建 Windows 应用程序。

### 33.1.2 Hello, Windows

下面开始使用 Visual Studio 2017 创建一个新的 Windows 应用程序。选择 Universal Windows Platform 项目。要回答的第一个问题是指定要支持的目标平台和最小平台(参见图 33-1)。在每一个更新的平台版本中，都有更多的特性。然而，需要注意用户有什么版本的 Windows 10。如果不支持平台版本，他们就无法安装和运行 Windows 10 应用程序。

对于所选择的目标版本，指定应用程序可以使用的 API 版本。对于最小版本，指定安装和运行应用程序的构建版本。如果将目标版本和最小版本设置为不同的值，就需要编写自适应代码。在调用 API 之前，需要确保 API 在受支持的构建版本中可用。如果没有新的 API，可以减少应用程序的特性，或者根据用户正在运行的构建版本提供不同的特性。

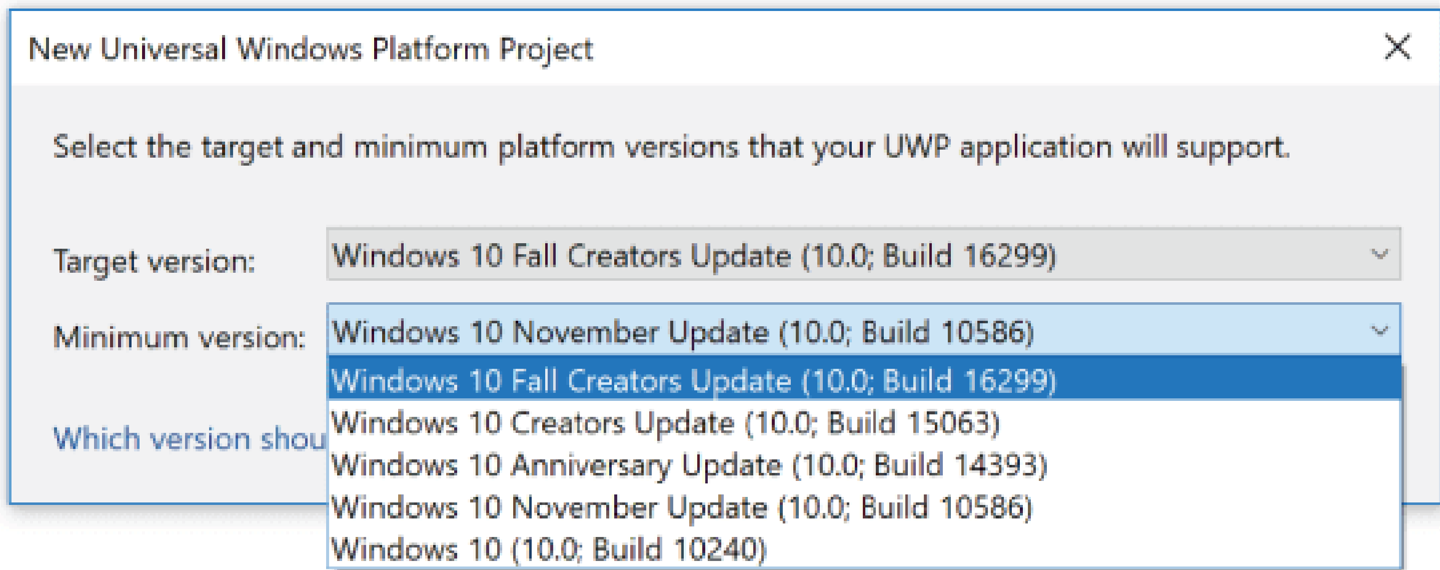



图 33-1

表 33-1 列出了 Windows 10 版本、构建号、发布日期和构建版本的特性。在 API 文档中，有时需要版本号、有时需要构建号来了解是否有 API 满足支持需求。

表 33-1

| 版本号 | 构建号 | 构 建 名 | 发 布 日 期 | 特　　性 |
|---|---|---|---|---|
| 1507 | 10240 | | 2015.7 | 更好的自适应布局、已编译的数据绑定、列表的声明式增量呈现(阶段)、延迟 UI 加载、RelativePanel、SplitPanel、CalendarDatePicker、Cortana 以及应用程序之间的通信 |
| 1511 | 10586 | November Update | 2015.11 | Windows Hello、Storage API 更新以及用于实时音频/视频调用的 ORTC(Object Real-Time Communication，对象实时通信) |
| 1607 | 14393 | Anniversary Update | 2016.7 | 新的墨水控件、Cortana API、带有 XAML 图像的动画 GIF、合成交互以及连接的动画 |
| 1703 | 15063 | Creators Update | 2017.3 | 新的合成 API，对量角器和尺子的油墨支持，地图上的图像以及 RadialController (表盘) |
| 1709 | 16299 | Fall Creators Update | 2017.10 | .NET Standard 2.0、流畅的设计、有条件的 XAML、用户活动、My People 以及新的 UI 控件(ColorPicker、NavigationView、PersonPicture、RatingControl) |

**注意：**

要使用.NET Standard 2.0 库，应用程序需要构建版本 16299 作为最小构建号。本书的大多数示例应用程序都将目标版本和最小版本设置为 16299。

### 33.1.3　应用程序清单文件

可以使用项目属性更改构建目标和最小版本号。Windows 应用程序还有另一个重要的打包配置。文件 Package.appxmanifest 可以使用 Package Manifest Editor 配置。

在 Application 设置(参见图 33-2)中，可以配置应用程序的显示名称、默认语言、支持设备的旋转以及定期的自动块更新。

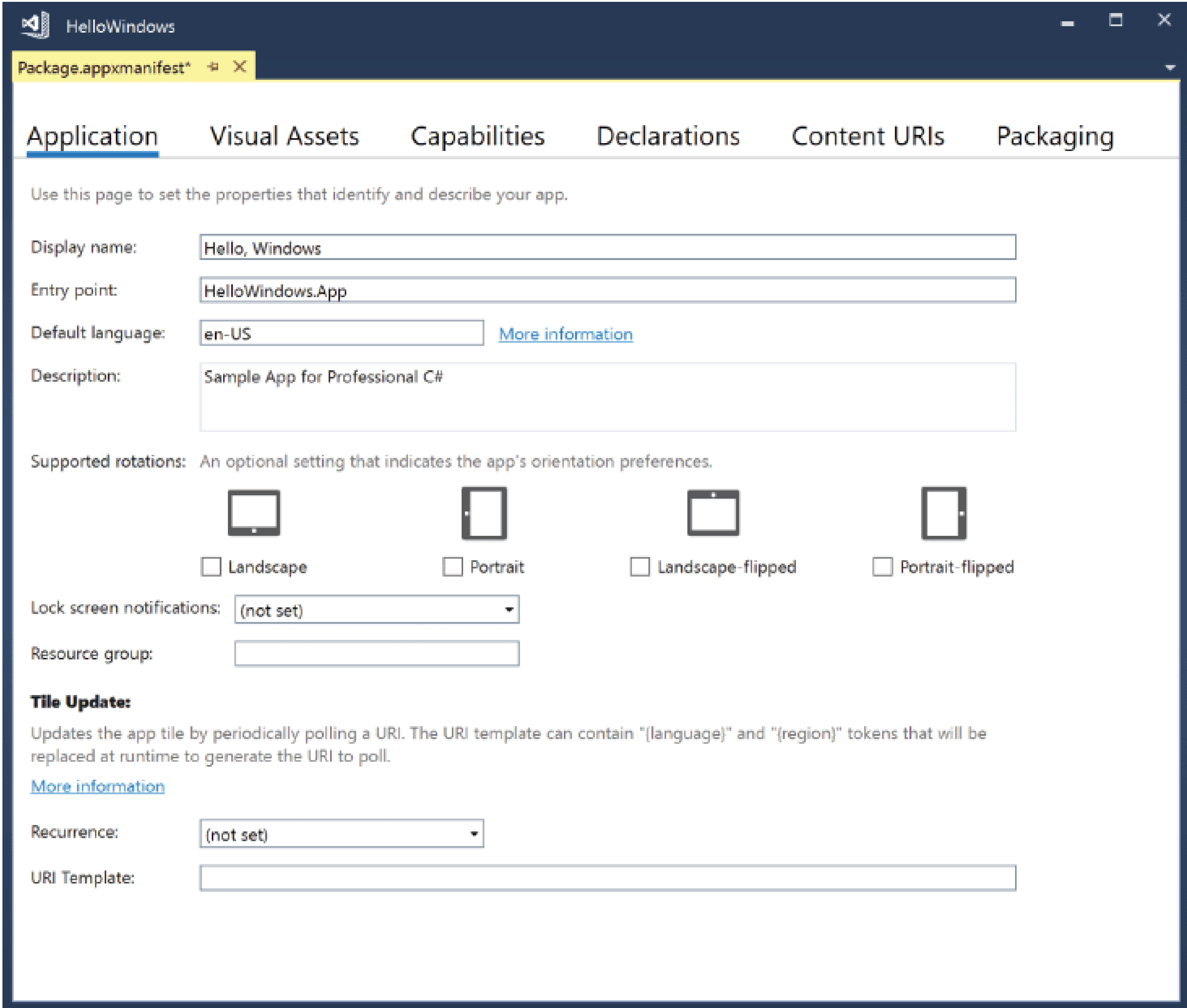



图 33-2

在 Visual Assets 选项卡中，可以配置应用程序的所有不同图标：给不同的磁贴大小、不同的设备分辨率、闪屏和 Windows Store 的包标识指定磁贴图像。

Capabilities 选项卡中的设置(参见图 33-3)允许选择应用程序所需的功能。默认选择的功能是 Internet(客户端)功能。如果需要对服务器执行网络请求，则需要打开此功能。其他功能允许访问位置信息，访问麦克风和摄像头，访问文件系统上的不同文件夹，如音乐库、图片库或视频库。

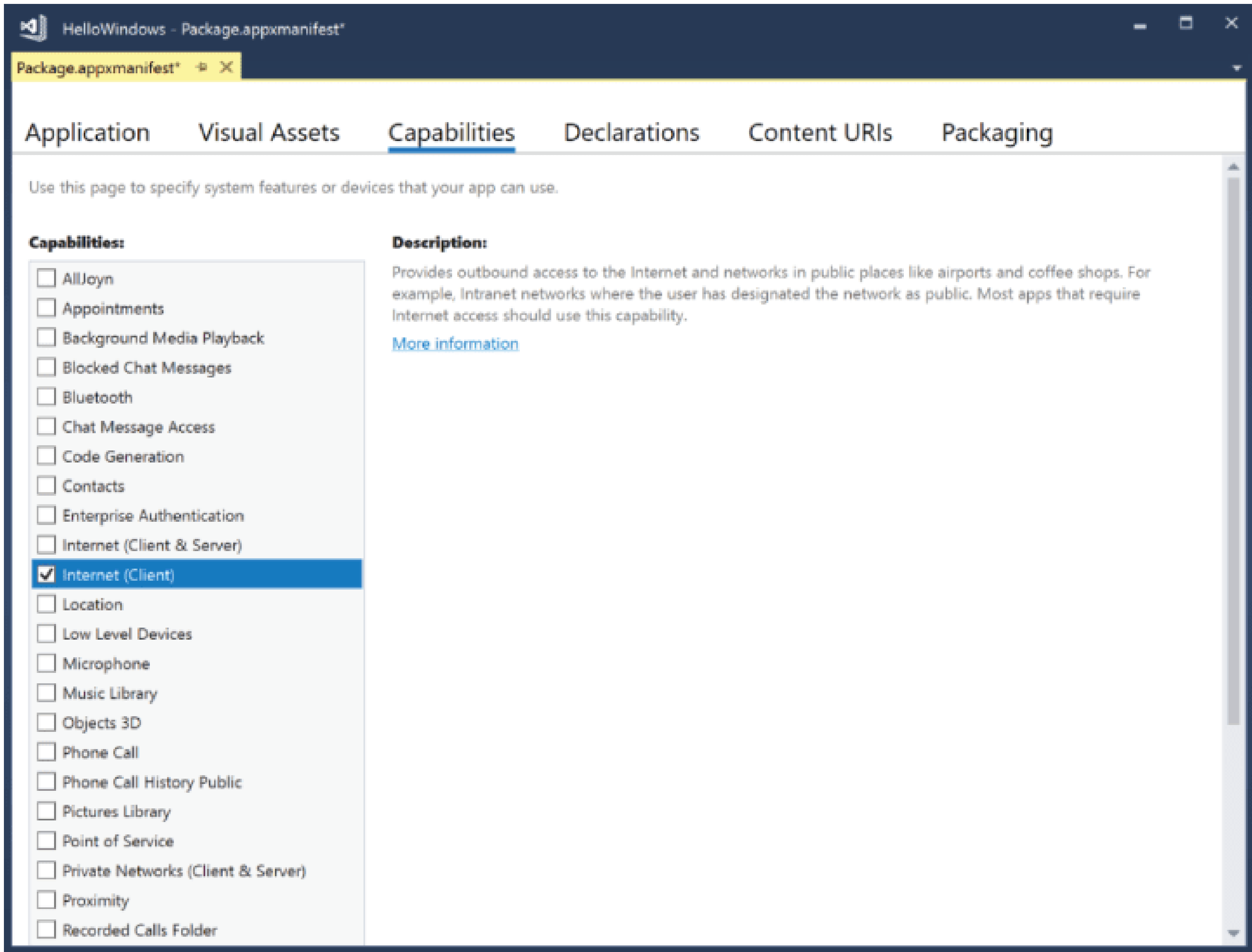



图 33-3

在 Declarations 设置(参见图 33-4)中，可以添加 Windows 需要了解的应用程序特性。例如，当共享一个应用程序的数据时，Windows 显示接受共享数据的应用程序。为此，应用程序需要注册为 Share Target。该选项卡中包含将应用程序注册为共享目标，通过协议或文件类型扩展来激活，在应用程序服务之间进行通信，以及更多的声明设置。

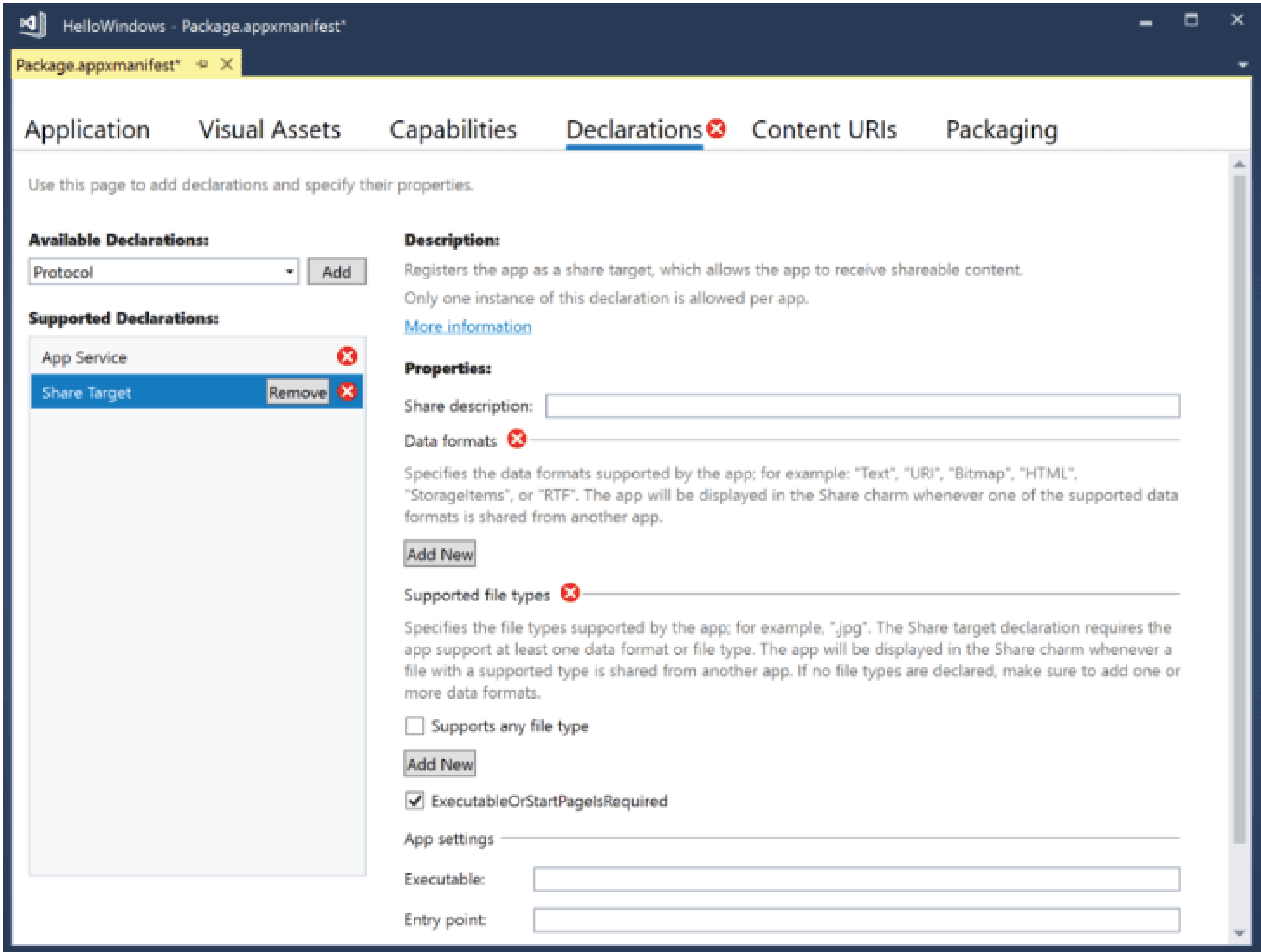



图 33-4

Content URIs 选项卡允许在应用程序中进行深度链接。在这里，可以指定在应用程序中打开页面的 URL。最后，使用 Packaging 选项卡，可以配置包名、版本和关于发布者的信息。

**注意：**
本书中讨论了一些功能和声明。请阅读第 36 章，获得有关此类功能的更多信息。

### 33.1.4　应用程序启动

应用程序的入口点 HelloWindows.App 在应用程序清单中定义，如前一节所示。App 类派生自 Application 基类，包含处理 Suspending 事件的实现代码(代码文件 HelloWindows/App.xam.cs)：

```
sealed partial class App : Application
{
  public App()
  {
    this.InitializeComponent();
    this.Suspending += OnSuspending;
  }
  //...
```

OnLaunched 方法被重写。当用户启动应用程序时，将调用此方法。如应用程序清单所述，启动应用程序有不同的方式。应用程序可以直接由用户启动，也可以在用户从另一个应用程序共享数据时启动。对于此类场景，可以重写不同的激活方法。

当用户启动应用程序时，也存在不同的场景。可以重新启动应用程序，或者，如果应用程序以前被挂起，则可以重新启动。当应用程序启动时，没有当前的 Frame，就会创建一个新的 Frame。Frame 可以用于页面之间的导航。页面包含 UI 控件，Frame 提供页面之间的导航。实例化 Frame 后，通过将 Navigate 方法调用到 MainPage，进行导航(代码文件 HelloWindows/App.xaml.cs)：

```
protected override void OnLaunched(LaunchActivatedEventArgs e)
{
  Frame rootFrame = Window.Current.Content as Frame;

  if (rootFrame == null)
  {
    rootFrame = new Frame();

    rootFrame.NavigationFailed += OnNavigationFailed;

    if (e.PreviousExecutionState == ApplicationExecutionState.Terminated)
    {
      //TODO: Load state from previously suspended application
    }

    Window.Current.Content = rootFrame;
  }

  if (e.PrelaunchActivated == false)
  {
    if (rootFrame.Content == null)
    {
      rootFrame.Navigate(typeof(MainPage), e.Arguments);
    }
    Window.Current.Activate();
  }
}
```

### 33.1.5　主页

下面向 MainPage 添加一个带有消息的按钮。用户界面使用 XAML((eXtensible Application Markup Language，可扩展应用程序标记语言)定义，这是一种用某些功能扩展 XML 的语言。对于按钮控件，Content 设置为显示一个简单的字符串，Click 事件分配给 OnButtonClicked 事件处理程序，属性 HorizontalAlignment、VerticalAlignment 和 Margin 用来使按钮占满网格中除按钮页边距之外的全部空间(代码文件 HelloWindows/MainPage.xaml)：

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
  <Button
    Content="Hello, Windows!"
    Click="OnButtonClicked"
    HorizontalAlignment="Stretch"
    VerticalAlignment="Stretch"
    Margin="20" />
</Grid>
```

在代码隐藏文件中，MessageDialog 使用 OnButtonClicked 事件处理程序显示。为了不阻塞用户界面，ShowAsync 方法只能作为异步方法使用。

```
private async void OnButtonClicked(object sender, RoutedEventArgs e)
{
  await new MessageDialog("Hello, Windows!").ShowAsync();
}
```

构建、部署和运行应用程序时，用户界面如图 33-5 所示。Windows 应用程序不仅需要在运行之前构建，还需要部署。在 Configuration Manager (Build | Configuration Manager) (参见图 33-6)中设置 Deploy 配置时，就可以在 Visual Studio 构建时自动完成部署。如果在构建时没有部署应用程序，就需要在构建后部署它，方法是在 Solution Explorer 中选择项目，在上下文菜单中使用 Deploy 选项；或者可以选择 Build | Deploy Solution，使用 Visual Studio 部署解决方案中的所有项目。

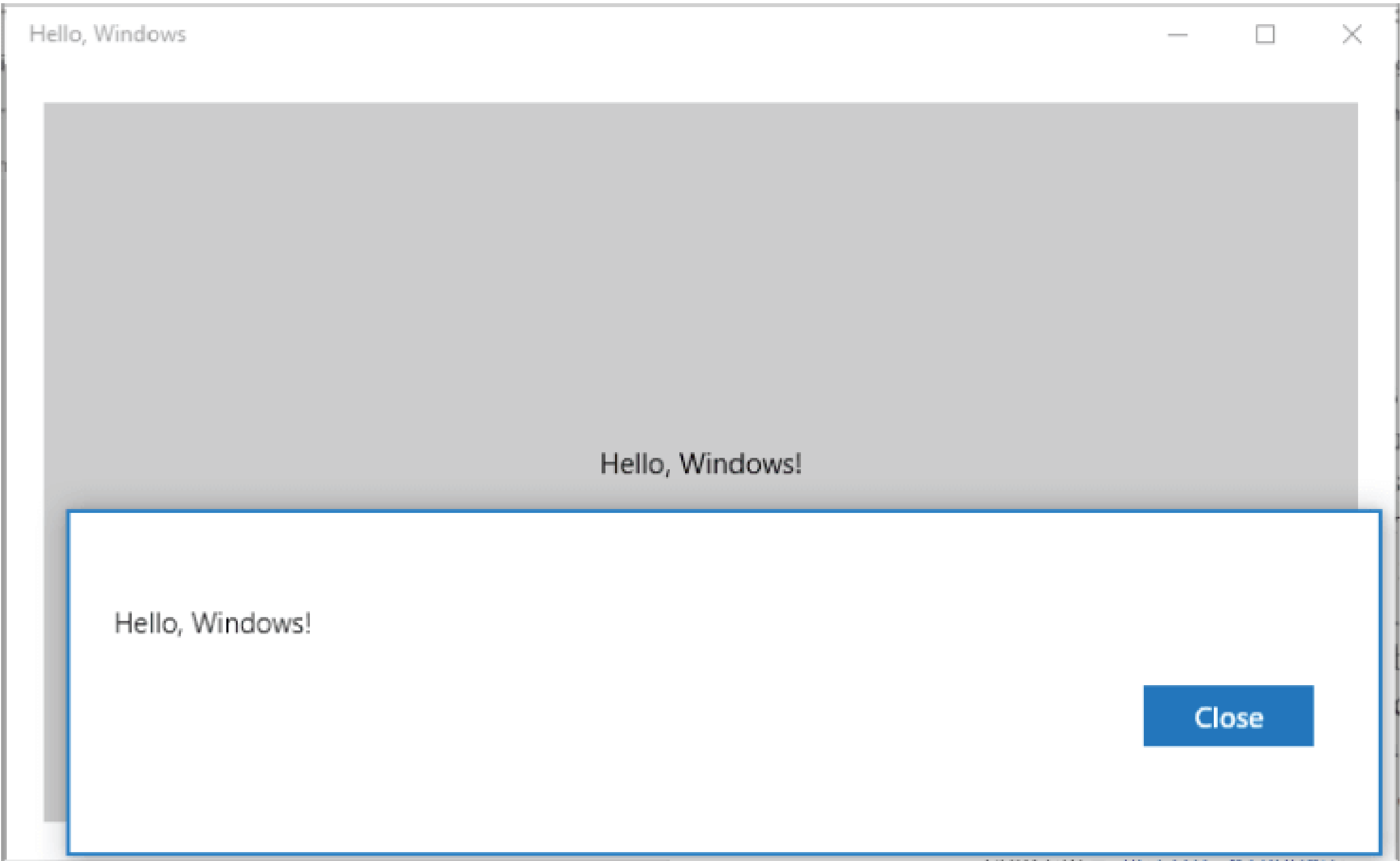



图 33-5

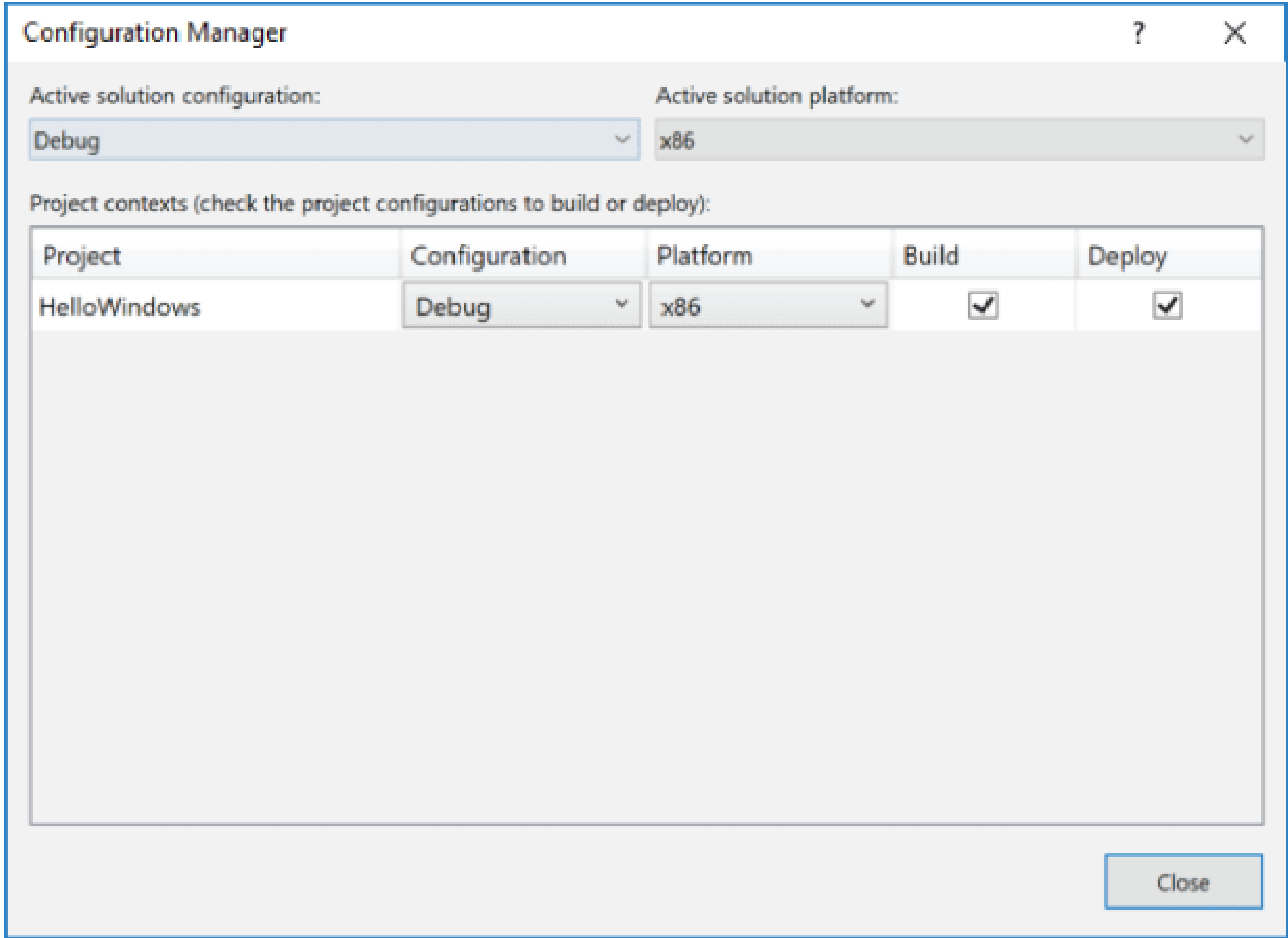



图 33-6

## 33.2 XAML

用 ASP.NET Core 编写 Web 应用程序时，除了需要知道 C#之外，还需要了解 HTML、CSS 和 JavaScript。创建 Windows 应用程序时，除了 C#之外，还需要了解 XAML。XAML 不仅用于创建 Windows 应用程序，还用于 Windows Presentation Foundation (WPF)、Windows Workflow Foundation (WF)和 Xamarin 的跨平台应用程序。

可以用 XAML 完成的工作都可以用 C#实现，每个 XAML 元素都用一个类表示，因此可以从 C#中访问。那么，为什么还需要 XAML？XAML 通常用于描述对象及其属性，可以描述很深的层次结构。例如，Page 包含一个 Grid 控件，Grid 控件包含一个 StackPanel 和其他控件，StackPanel 包含按钮和文本框控件。XAML 便于描述这种层次结构，并通过 XML 特性或元素分配对象的属性。

XAML 允许以声明的方式编写代码，而 C#主要是一种命令式编程语言。XAML 支持声明式定义。在命令式编程语言(如 C#)中，用 C#代码定义一个 for 循环，编译器就使用中间语言(IL)代码创建一个 for 循环。在声明性编程语言中，声明应该做什么，而不是如何完成。

**注意：**

虽然 C#不是纯粹的命令式编程语言，但使用 LINQ 时，也是在以声明方式编写语法。Entity Framework Core (EF Core)的 LINQ 提供程序将 LINQ 查询转换为 SQL 语句。参见第 26 章。

XAML 是一个 XML 语法，但它定义了 XML 的几个增强特性。XAML 仍然是有效的 XML，但是一些增强特性有特殊的意义和特殊的功能，例如，在 XML 特性中使用花括号，对于 XML，这仍然只是一个字符串，因此是有效的 XML。对于 XAML，这是一个标记扩展。

在有效使用 XAML 之前，需要了解这门语言的一些重要特性。本章介绍了如下 XAML 特性：

- **依赖属性：**从外部看起来，依赖属性像正常属性。然而，它们需要更少的存储空间，实现了变更通知。
- **路由事件：**从外部看起来，路由事件像正常的.NET 事件。然而，通过添加和删除访问器来使用自定义事件实现方式，就允许冒泡和隧道。事件从外部控件进入内部控件称为隧道，从内部控件进入外部控件称为冒泡。
- **附加属性：**通过附加属性，可以给其他控件添加属性。例如，按钮控件没有属性用于把它自己定位在 Grid 控件的特定行和列上。在 XAML 中，看起来有这样一个属性。
- **标记扩展：**编写 XML 特性需要的编码比编写 XML 元素少。然而，XML 特性只能是字符串；使用 XML 元素可以编写更强大的语法。为了减少需要编写的代码量，标记扩展允许在特性中编写强大的语法。

**注意：**

.NET 属性参见第 3 章。事件，包括通过添加和删除访问器编写自定义事件，详见第 8 章。XML 的功能参见网上附加第 2 章。

### 33.2.1 XAML 标准

WPF、UWP 和 Xamarin 对 XAML 元素使用(部分仍然使用)不同的语法。例如，对于 WPF 和 UWP，按钮有 Content 属性，而 Xamarin 的按钮有 Text 属性。在 WPF 和 UWP 中，可以使用 StackPanel 来排列多个元素。在 Xamarin 中，类似的控件是 StackLayout。

为了更容易地在不同的 UI 技术堆栈之间切换，定义了 XAML 标准。有关标准的实际状态，请参见 https://github.com/Microsoft/xaml-standard/。

### 33.2.2 将元素映射到类

在每个 XAML 元素的后面都有一个具有属性、方法和事件的类。如前所述，可以使用 C#代码或使用 XAML 创建 UI 元素。下面看一个例子。使用以下代码片段，定义了一个包含按钮控件的 StackPanel。使用 XML 特性，

按钮分配了 Content 属性和 Click 事件。Content 属性只包含一个简单的字符串，而 Click 事件引用了方法 OnButtonClick 的地址。XML 特性 x:Name 用于向按钮控件声明一个名称，该名称可以在 XAML 和 C#代码隐藏文件中使用(代码文件 XAMLIntro/MainPage.xaml)：

```
<StackPanel x:Name="stackPanel1">
  <Button Content="Click Me!" x:Name="button1" Click="OnButtonClick" />
  <!-- ... -->
</StackPanel>
```

在页面顶部，可以看到带有 XML 特性 x:Class 的 Page 元素。这定义了类的名称，在该类中，XAML 编译器生成了部分代码。使用 Visual Studio 中的代码隐藏文件，可以看到这个类中能修改的部分(代码文件 XAMLIntro/MainPage.xaml)：

```
<Page
  x:Class="XAMLIntro.MainPage"
  <!-- ... -->
```

代码隐藏文件包含类 MainPage 的一部分(XAML 编译器没有生成这个部分)。在构造函数中，调用方法 InitializeComponent。InitializeComponent 的实现是由 XAML 编译器创建的。该方法加载 XAML 文件，并将其转换为 XAML 文件中的根元素指定的对象。OnButtonClick 方法是之前在 XAML 代码中创建的按钮的 Click 事件处理程序。这个实现打开了一个 MessageDialog(代码文件 XAMLIntro/MainPage.xam.cs)：

```
public sealed partial class MainPage : Page
{
  public MainPage()
  {
    this.InitializeComponent();
  }

  private async void OnButtonClick(object sender, RoutedEventArgs e)
  {
    await new MessageDialog("button 1 clicked").ShowAsync();
  }
}
```

现在，在 C#代码的 Button 类中创建一个新对象，并将其添加到现有的 StackPanel 中。在下面的代码片段中，修改了 MainPage 的构造函数，以创建一个新按钮，设置 Content 属性，并为 Click 事件分配一个 Lambda 表达式。最后，新创建的按钮添加到 StackPanel 的 Children 中(代码文件 XAMLIntro/mainpage.xaml.cs)：

```
public MainPage()
{
  this.InitializeComponent();
  var button2 = new Button
  {
    Content = "created dynamically"
  };
  button2.Click += async (sender, e) =>
    await new MessageDialog("button 2 clicked").ShowAsync();
  stackPanel1.Children.Add(button2);
}
```

如前所述，XAML 只是处理对象、属性和事件的另一种方式。下一节将展示 XAML 在用户界面上的优势。

### 33.2.3 通过 XAML 使用定制的.NET 类

要在 XAML 代码中使用自定义的.NET 类，可以使用简单的 POCO 类，对类定义没有特殊要求。只需要将.NET 名称空间添加到 XAML 声明中。为了演示这一点，下面设计一个具有 FirstName 和 LastName 属性的简单 Person 类(代码文件 DataLib/Person.cs)：

```
public class Person
{
  public string FirstName { get; set; }
  public string LastName { get; set; }
  public override string ToString() => $"{FirstName} {LastName}";
}
```

在 XAML 中定义了一个名为 datalib 的 XML 名称空间别名，它映射到程序集 DataLib 中的.NET 名称空间

DataLib。有了这个别名，现在就可以把别名作为元素的前缀，来使用这个名称空间中的所有类。

在 XAML 代码中添加一个列表框，其中包含 Person 类型的项。使用 XAML 特性，可以设置属性 FirstName 和 LastName 的值。运行应用程序时，ToString 方法的输出显示在列表框中(代码文件 XAMLIntro/MainPage.xaml)：

```
<Page x:Class="XamlIntro.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:XAMLIntro"
  xmlns:datalib="using:DataLib"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">
  <StackPanel x:Name="stackPanel1" >
    <Button Content="Click Me!" x:Name="button1" Click="OnButtonClick">
    <ListBox>
      <datalib:Person FirstName="Stephanie" LastName="Nagel" />
      <datalib:Person FirstName="Matthias" LastName="Nagel" />
      <datalib:Person FirstName="Katharina" LastName="Nagel" />
    </ListBox>
  </StackPanel>
</Window>
```

**注意：**

WPF 和 Xamarin 在别名声明中使用 clr-namespace 而不是使用 using。原因是，使用 UWP 的 XAML 既不基于.NET，也不局限于.NET。可以使用本机 C++和 XAML，因此 clr(公共语言运行库)就不适合了。

### 33.2.4　将属性用作特性

在前面的 XAML 示例中，类的属性用 XML 特性来设置。要在 XAML 中设置属性，只要属性的类型可以表示为字符串，或者可以把字符串转换为属性类型，就可以把属性设置为特性。下面的代码片段用 XML 特性设置了 Button 元素的 Content 和 Background 属性。

```
<Button Content="Click Me!" Background="LightGoldenrodYellow" />
```

在上面的代码片段中，因为 Content 属性的类型是 object，所以可以接受字符串。Background 属性的类型是 Brush，字符串转换为派生自 Brush 的 SolidColorBrush 类型。

### 33.2.5　将属性用作元素

总是可以使用元素语法给属性提供值。Button 类的 Background 属性可以用子元素 Button. Background 设置。下面的代码用特性定义了 Button，效果是相同的：

```
<Button>
  Click Me!
  <Button.Background>
    <SolidColorBrush Color="LightGoldenrodYellow" />
  </Button.Background>
</Button>
```

使用元素代替特性，可以把比较复杂的画笔应用于 Background 属性(如 LinearGradientBrush)，如下面的示例所示(代码文件 XAMLIntro/MainWindow.xaml)。

```
<Button x:Name="button1" Click="OnButtonClick">
  Click Me!
  <Button.Background>
    <LinearGradientBrush StartPoint="0.5,0.0" EndPoint="0.5, 1.0">
      <GradientStop Offset="0" Color="Yellow" />
      <GradientStop Offset="0.3" Color="Orange" />
      <GradientStop Offset="0.7" Color="Red" />
      <GradientStop Offset="1" Color="DarkRed" />
    </LinearGradientBrush>
  </Button.Background>
</Button>
```

**注意：**

当设置示例中的内容时，Content 特性和 Button.Content 元素都不用于编写内容；相反，内容会直接写入为 Button 元素的子元素值。这是因为在 Button 类的基类 ContentControl 中，ContentProperty 特性通过[ContentProperty("Content")]应用。这个特性把 Content 属性标记为 ContentProperty。这样，XAML 元素的直接子元素就应用于 Content 属性。

### 33.2.6 依赖属性

XAML 使用依赖属性完成数据绑定、动画、属性变更通知、样式化等。依赖属性存在的原因是什么？假设创建一个类，它有 100 个 int 型的属性，这个类在一个表单上实例化了 100 次。需要多少内存？因为 int 的大小是 4 个字节，所以结果是 4×100×100 = 40 000 字节。刚才看到的是一个 XAML 元素的属性？由于继承层次结构非常大，一个 XAML 元素定义了数以百计的属性。属性类型不是简单的 int，而是更复杂的类型。这样的属性会消耗大量的内存。然而，通常只改变其中一些属性的值，大部分的属性保持对所有实例都相同的默认值。这个难题可以用依赖属性解决。使用依赖属性，对象内存不是分配给每个属性和实例。依赖属性系统管理一个包含所有属性的字典，只有值发生了改变才分配内存。否则，默认值就在所有实例之间共享。

依赖属性也内置了对变更通知的支持。对于普通属性，需要为变更通知实现 INotifyProperty Changed 接口。其方式参见本章的“数据绑定”一节。这种变更机制是通过依赖属性内置的。对于数据绑定，绑定到.NET 属性源上的 UI 元素的属性必须是依赖属性。现在，详细讨论依赖属性。

从外部来看，依赖属性像是正常的.NET 属性。但是，正常的.NET 属性通常还定义了由该属性的 get 和 set 访问器访问的数据成员。

```
private int _value;
public int Value
{
  get => _value;
  Set => _value = value;
}
```

依赖属性不是这样。依赖属性通常也有 get 和 set 访问器。它们与普通属性是相同的。但在 get 和 set 访问器的实现代码中，调用了 GetValue()和 SetValue()方法。GetValue()和 SetValue()方法是基类 DependencyObject 的成员，依赖对象需要使用这个类——它们必须在 DependencyObject 的派生类中实现。

有了依赖属性，数据成员就放在由基类管理的内部集合中，仅在值发生变化时分配数据。对于没有变化的值，数据可以在不同的实例或基类之间共享。GetValue()和 SetValue()方法需要一个 DependencyProperty 参数。这个参数由类的一个静态成员定义，该静态成员与属性同名，并在该属性名的后面追加 Property 术语。对于 Value 属性，静态成员的名称是 ValueProperty。DependencyProperty.Register()是一个辅助方法，可在依赖属性系统中注册属性。在下面的代码片段中，使用 Register()方法和 4 个参数定义了属性名、属性的类型和拥有者的类型(即 MyDependencyObject 类)，使用 PropertyMetadata 指定了默认值(代码文件 DependencyObjectSample/MyDependencyObject.cs)。

```
public class MyDependencyObject: DependencyObject
{
  public int Value
  {
    get => (int)GetValue(ValueProperty);
    set => SetValue(ValueProperty, value);
  }

  public static readonly DependencyProperty ValueProperty =
    DependencyProperty.Register("Value", typeof(int),
      typeof(MyDependencyObject), new PropertyMetadata(0));
}
```

### 33.2.7 创建依赖属性

下面的示例定义的不是一个依赖属性，而是 3 个依赖属性。MyDependencyObject 类定义了依赖属性 Value、

Minimum 和 Maximum。所有这些属性都是用 DependencyProperty.Register()方法注册的依赖属性。GetValue()和 SetValue()方法是基类 DependencyObject 的成员。对于 Minimum 和 Maximum 属性，定义了默认值，用 DependencyProperty.Register()方法设置该默认值时，可以把第 4 个参数设置为 PropertyMetadata。使用带一个参数 PropertyMetadata 的构造函数，把 Minimum 属性设置为 0，把 Maximum 属性设置为 100(代码文件 DependencyObjectSample/MyDependencyObject.cs)。

```
public class MyDependencyObject: DependencyObject
{
  public int Value
  {
    get => (int)GetValue(ValueProperty);
    set => SetValue(ValueProperty, value);
  }

  public static readonly DependencyProperty ValueProperty =
    DependencyProperty.Register(nameof(Value), typeof(int),
      typeof(MyDependencyObject));

  public int Minimum
  {
    get => (int)GetValue(MinimumProperty);
    set => SetValue(MinimumProperty, value);
  }

  public static readonly DependencyProperty MinimumProperty =
    DependencyProperty.Register(nameof(Minimum), typeof(int),
      typeof(MyDependencyObject), new PropertyMetadata(0));

  public int Maximum
  {
    get => (int)GetValue(MaximumProperty);
    set => SetValue(MaximumProperty, value);
  }

  public static readonly DependencyProperty MaximumProperty =
    DependencyProperty.Register(nameof(Maximum), typeof(int),
      typeof(MyDependencyObject), new PropertyMetadata(100));
}
```

**注意：**

在 get 和 set 属性访问器的实现代码中，只能调用 GetValue()和 SetValue()方法。使用依赖属性，可以通过 GetValue()和 SetValue()方法从外部访问属性的值，UWP 也是这样做的；因此，强类型化的属性访问器可能根本就不会被调用，包含它们仅为了方便在自定义代码中使用正常的属性语法。

## 33.2.8　值变更回调和事件

为了获得值变更的信息，依赖属性还支持值变更回调。在属性值发生变化时调用的 DependencyProperty.Register()方法中，可以添加一个 DependencyPropertyChanged 事件处理程序。在示例代码中，把 OnValueChanged()处理程序方法赋予 PropertyMetadata 对象的 PropertyChangedCallback 属性。在 OnValueChanged()方法中，可以用 DependencyPropertyChangedEventArgs()参数访问属性的新旧值(代码文件 DependencyObjectSample/MyDependencyObject.cs)。

```
public class MyDependencyObject: DependencyObject
{
  public int Value
  {
    get => (int)GetValue(ValueProperty);
    set => SetValue(ValueProperty, value);
  }

  public static readonly DependencyProperty ValueProperty =
    DependencyProperty.Register(nameof(Value), typeof(int),
      typeof(MyDependencyObject),
      new PropertyMetadata(0, OnValueChanged, CoerceValue));

  private static void OnValueChanged(DependencyObject obj,
```

```
        DependencyPropertyChangedEventArgs e)
      {
        int oldValue = (int)e.OldValue;
        int newValue = (int)e.NewValue;
        //...
      }
    }
```

## 33.2.9 路由事件

第 8 章介绍了.NET 事件模型。使用默认实现的事件，当触发事件时，将调用直接连接到事件的处理程序。使用 UI 技术时，对事件处理有不同的需求。在一些事件中，应该可以创建一个带有容器控件的处理程序，并对来自子控件的事件做出反应。这可以通过为.NET 事件创建自定义实现代码来实现，如第 8 章的 add 和 remove 访问器所示。

UWP 提供了路由事件。示例应用程序定义的用户界面包含一个复选框，如果选中它，就停止路由；一个按钮控件，其 Tapped 事件设置为 OnTappedButton 处理程序方法；一个网格，其 Tapped 事件设置为 OnTappedGrid 处理程序。Tapped 事件是 Universal Windows 应用程序的一个路由事件。这个事件可以用鼠标、触摸屏和笔设备触发(代码文件 RoutedEvents/MainPage.xaml)：

```
<Grid Tapped="OnTappedGrid">
  <Grid.RowDefinitions>
    <RowDefinition Height="auto" />
    <RowDefinition Height="auto" />
    <RowDefinition />
  </Grid.RowDefinitions>
  <StackPanel Grid.Row="0" Orientation="Horizontal">
    <CheckBox x:Name="CheckStopRouting">Stop Routing</CheckBox>
    <Button Click="OnCleanStatus">Clean Status</Button>
  </StackPanel>
  <Button Grid.Row="1" Tapped="OnTappedButton">Tap me!</Button>
  <TextBlock Grid.Row="2" Margin="20" x:Name="textStatus" />
</Grid>
```

OnTappedXX 处理程序方法把状态信息写入一个 TextBlock，来显示处理程序方法和事件初始源的控件 (代码文件 RoutedEventsUWP/MainPage.xaml.cs)：

```
private void OnTappedButton(object sender, TappedRoutedEventArgs e)
{
  ShowStatus(nameof(OnTappedButton), e);
  e.Handled = CheckStopRouting.IsChecked == true;
}

private void OnTappedGrid(object sender, TappedRoutedEventArgs e)
{
  ShowStatus(nameof(OnTappedGrid), e);
  e.Handled = CheckStopRouting.IsChecked == true;
}

private void ShowStatus(string status, RoutedEventArgs e)
{
  textStatus.Text += $"{status} {e.OriginalSource.GetType().Name}";
  textStatus.Text += "\r\n";
}

private void OnCleanStatus(object sender, RoutedEventArgs e)
{
  textStatus.Text = string.Empty;
}
```

运行应用程序，在网格内单击按钮的外部，就会看到处理的 OnTappedGrid 事件，并把 Grid 控件作为触发事件的源：

```
OnTappedGrid Grid
```

单击按钮的中间，会看到事件被路由。第一个调用的处理程序是 OnTappedButton，其后是 OnTappedGrid：

```
OnTappedButton TextBlock
OnTappedGrid TextBlock
```

同样有趣的是，事件源不是按钮，而是 TextBlock。原因在于，这个按钮使用 TextBlock 设置样式，来包含

按钮的文本。如果单击按钮内的其他位置，还可以看到 Grid 或 ContentPresenter 是原始事件源。Grid 和 ContentPresenter 是创建按钮的其他控件。

在单击按钮之前，选中复选框 CheckStopRouting，可以看到事件不再路由，因为事件参数的 Handled 属性被设置为 true：

```
OnTappedButton TextBlock
```

在事件的 Microsoft API 文档内，可以在文档的备注部分看到事件类型是否路由。在 Universal Windows 应用程序中，tapped、drag 和 drop、key up 和 key down、pointer、focus、manipulation 事件是路由事件。

## 33.2.10　附加属性

依赖属性是可用于特定类型的属性。而通过附加属性，可以为其他类型定义属性。一些容器控件为其子控件定义了附加属性；例如，如果使用 DockPanel 控件，就可以为其子控件使用 Dock 属性。Grid 控件定义了 Row 和 Column 属性。

下面的代码片段说明了附加属性在 XAML 中的情况。Button 类没有 Grid.Dock 属性，但它是从 Grid 控件附加的。

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition />
    <RowDefinition />
  </Grid.RowDefinitions>
  <Button Content="First" Grid.Row="0" Background="Yellow" />
  <Button Content="Second" Grid.Row="1" Background="Blue" />
</Grid>
```

附加属性的定义与依赖属性非常类似，如下面的示例所示。定义附加属性的类必须派生自基类 DependencyObject，并定义一个普通的属性，其中 get 和 set 访问器调用基类的 GetValue()和 SetValue()方法。这些都是类似之处。接着不调用 DependencyProperty 类的 Register()方法，而是调用 RegisterAttached()方法。RegisterAttached()方法注册一个附加属性，现在它可用于每个元素(代码文件 AttachedProperty/MyAttached PropertyProvider.cs)。

```
public class MyAttachedPropertyProvider: DependencyObject
{
  public static readonly DependencyProperty MySampleProperty =
    DependencyProperty.RegisterAttached
      "MySample",
      typeof(string),
      typeof(MyAttachedPropertyProvider),
      new PropertyMetadata(string.Empty));

  public static void SetMySample(UIElement element, string value) =>
    element.SetValue(MySampleProperty, value);

  public static int GetMyProperty(UIElement element) =>
    (string)element.GetValue(MySampleProperty);
}
```

**注意：**

似乎 Grid.Row 属性只能添加到 Grid 控件中的元素。实际上，附加属性可以添加到任何元素上。但无法使用这个属性值。Grid 控件能够识别这个属性，并从其子元素中读取它，以安排其子元素。它不从子元素的子元素中读取。

在 XAML 代码中，附加属性现在可以附加到任何元素上。第二个 Button 控件 button2 为自身附加了属性 MyAttachedPropertyProvider.MySample，其值指定为 42(代码文件 AttachedProperty/MainPage.xaml)。

```
<Grid x:Name="grid1">
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
    <RowDefinition Height="*" />
  </Grid.RowDefinitions>
  <Button Grid.Row="0" x:Name="button1" Content="Button 1" />
```

```
  <Button Grid.Row="1" x:Name="button2" Content="Button 2"
    local:MyAttachedPropertyProvider.MySample="42" />
  <ListBox Grid.Row="2" x:Name="list1" />
</Grid>
```

在代码隐藏中执行相同的操作时，必须调用 MyAttachedPropertyProvider 类的静态方法 SetMy Property()。不能扩展 Button 类，使其包含某个属性。SetProperty()方法获取一个应由该属性及其值扩展的 UIElement 实例。在如下的代码片段中，把该属性附加到 button1 中，将其值设置为 sample value(代码文件 AttachedProperty/MainPage.xaml.cs)。

```
public MainPage()
{
  InitializeComponent();
  MyAttachedPropertyProvider.SetMySample(button1, "sample value");
  //...
}
```

为了读取分配给元素的附加属性，可以使用 VisualTreeHelper 迭代层次结构中的每个元素，并试图读取其附加属性。VisualTreeHelper 用于在运行期间读取元素的可见树。GetChildrenCount 方法返回子元素的数量。为了访问子元素，可以使用 GetChild 方法，通过第二个参数传递一个元素的索引，该方法返回元素。只有当元素的类型是 FrameworkElement(或派生于它)，且用 Func 参数传递的谓词返回 true 时，该方法的实现代码才返回元素(代码文件 AttachedProperty/MainPage.xaml.cs)。

```
private IEnumerable<FrameworkElement> GetChildren(FrameworkElement element,
  Func<FrameworkElement, bool> pred)
{
  int childrenCount = VisualTreeHelper.GetChildrenCount(rootElement);
  for (int i = 0; i < childrenCount; i++)
  {
    var child = VisualTreeHelper.GetChild(rootElement, i) as FrameworkElement;
    if (child != null && pred(child))
    {
      yield return child;
    }
  }
}
```

GetChildren 方法现在在页面的构造函数中用于把带有附加属性的所有元素添加到 ListBox 控件中(代码文件 AttachedProperty/MainPage.xaml.cs)：

```
public MainPage()
{
  InitializeComponent();
  MyAttachedPropertyProvider.SetMySample(button1, "sample value");
  foreach (var item in GetChildren(grid1, e =>
    MyAttachedPropertyProvider.GetMySample(e) != string.Empty))
  {
    list1.Items.Add(
      $"{item.Name}: {MyAttachedPropertyProvider.GetMySample(item)}");
  }
}
```

运行应用程序(WPF 或 UWP 应用程序)时，会看到列表框中的两个按钮控件与下述值：

```
button1: sample value
button2: 42
```

**注意：**
本章的“布局面板”一节展示了许多不同的附加属性和许多容器控件，如 Canvas、Grid 和 RelativePanel。

### 33.2.11　标记扩展

通过标记扩展，可以扩展 XAML 的元素或特性语法。如果 XML 特性包含花括号，就表示这是标记扩展的一个符号。特性的标记扩展常常用作简写记号，而不再使用元素。

这种标记扩展的示例是 StaticResourceExtension，它可查找资源。下面是带有 gradientBrush1 键的线性渐变笔刷的资源(代码文件 MarkupExtensions/MainPage.xaml)：

```
<Page.Resources>
  <LinearGradientBrush x:Key="gradientBrush1" StartPoint="0.5,0.0"
    EndPoint="0.5, 1.0">
    <GradientStop Offset="0" Color="Yellow" />
    <GradientStop Offset="0.3" Color="Orange" />
    <GradientStop Offset="0.7" Color="Red" />
    <GradientStop Offset="1" Color="DarkRed" />
  </LinearGradientBrush>
</Page.Resources>
```

使用 StaticResourceExtension，通过特性语法来设置 Button 的 Background 属性，就可以引用这个资源。特性语法通过花括号和没有 Extension 后缀的扩展类名来定义。

```
<Button Content="Test" Background="{StaticResource gradientBrush1}" />
```

Windows 应用程序不支持可用于 WPF 的所有标记扩展，只支持其中的一些。StaticResource 和 ThemeResource 参见第 35 章，绑定标记扩展 Binding 和 x:Bind 在本章的"数据绑定"一节讨论。从 Fall Creators Update (Windows 10 构建号 16299)开始，也支持自定义标记扩展的创建，如下所述。

### 33.2.12 自定义标记扩展

自定义标记扩展允许在 XAML 代码的花括号中添加自己的特性。可以创建自定义绑定、基于条件的评估或简单的计算器，如下一个示例所示。

Calculator 标记扩展允许使用加、减、乘、除操作计算两个值。标记扩展非常简单：类名包含 Extension 后缀，它派生自基类 MarkupExtension，重写了方法 ProvideValue。使用 ProvideValue，标记扩展返回分配给属性的值或对象(在其中定义了标记)。返回值的类型由 MarkupExtensionReturnType 特性定义。下面的代码片段显示了 Calculator 标记扩展的实现。这个扩展定义了可以设置的三个属性：X、Y 的属性，以及应用于 X 和 Y 的 Operation。Operation 用一个枚举来定义。在 ProvideValue 方法的实现中，对 X 和 Y 应用一个操作，返回结果(代码文件 CustomMarkupExtension/CalculatorExtension.cs)：

```
public enum Operation
{
  Add,
  Subtract,
  Multiply,
  Divide
}

[MarkupExtensionReturnType(ReturnType = typeof(string))]
public class CalculatorExtension : MarkupExtension
{
  public double X { get; set; }
  public double Y { get; set; }
  public Operation Operation { get; set; }

  protected override object ProvideValue()
  {
    double result = 0;
    switch (Operation)
    {
      case Operation.Add:
        result = X + Y;
        break;
      case Operation.Subtract:
        result = X - Y;
        break;
      case Operation.Multiply:
        result = X * Y;
        break;
      case Operation.Divide:
        result = X / Y;
        break;
      default:
        break;
    }
    return result.ToString();
  }
}
```

现在，Calculator 标记扩展可以与 XML 特性语法一起使用。在这里，初始化标记扩展，以设置属性。返回的字符串应用于 TextBlock 的 Text 属性(代码文件 CustomMarkupExtension/MainPage.xaml)：

```
<TextBlock Text="{local:Calculator Operation=Add, X=38, Y=4}" />
```

使用标记扩展语法，不使用名称 Extension。这个后缀会自动应用。当然，如果 CalculatorExtension 类仅仅用于将它实例化为 Text 属性的子元素，并设置扩展的属性，这就是不同的(代码文件 CustomMarkupExtension/MainPage.xaml)：

```
<TextBlock>
  <TextBlock.Text>
    <local:CalculatorExtension Operation="Multiply" X="7" Y="6" />
  </TextBlock.Text>
</TextBlock>
```

运行应用程序时，在所使用的两个操作中返回值 42。

### 33.2.13 条件 XAML

如果需要支持 Windows 10 的多个构建版本，但是仍然使用更新的特性，比如自定义标记扩展，就需要编写自适应代码，这样，在旧版本的 Windows 上调用新 API 时，不会导致应用程序崩溃。使用自适应代码时，需要在调用 API 之前验证它是否可用，并为旧版本的 Windows 提供替代方法。这样的条件代码也可以用 XAML 实现，只要支持的最小构建版本号是 15063。条件 XAML 不支持旧版本。

条件 XAML 是通过指定 XAML 名称空间来实现的，这些名称空间只能根据特定的条件提供。例如，只有给 Windows.Foundation.UniversalApiContract 传递版本 5 的 IsApiContractPresent 返回 true，才能使用名称空间别名 contract5。否则，XAML 编译器会忽略这个名称空间别名，不使用这个别名创建元素或特性。名称空间之后的问号“?”定义了别名有效的条件(XAML 文件 ConditionalXAML/MainPage.xaml)：

```
xmlns:contract5=
  "http://schemas.microsoft.com/winfx/2006/xaml/presentation?
    IsApiContractPresent(Windows.Foundation.UniversalApiContract, 5)"
```

表 33-2 中的 API 可以与条件 XAML 一起使用。使用 IsApiContractPresent 方法，可以检查特定版本号的特定 API 是否可用。如果指定的控件类型可用，则 IsTypePresent 返回 true；如果控件的特定属性可用，则 IsPropertyPresent 返回 true。逆方法也用所有这些方法来实现。逆方法返回反布尔值。例如，如果 IsApiContractPresent 返回 true，则方法 IsApiContractNotPresent 返回 false。那么，为什么逆方法是必需的，而不能仅仅使用 C#的 !操作符？原因是 XAML 没有这样的操作符。

表 33-2

| 方　法 | 逆 方 法 |
|---|---|
| IsApiContractPresent(contract, version) | IsApiContractNotPresent(contract, version) |
| IsTypePresent(type) | IsTypeNotPresent(type) |
| IsPropertyPresent(type, property) | IsPropertyNotPresent(type, property) |

现在，名称空间别名可以在 http://schemas.microsoft.com/winfx/2006/xaml/presentation 名称空间中使用元素和特性，例如，TextBlock 的 Text 属性：

```
<TextBlock x:Name=text1 contract5:Text="contract 5 present" />
```

使用 C#代码，可以使用名称空间 Windows.Foundation.Metadata 的 ApiInformation 类中定义的 IsApiContractPresent 方法来实现类似的功能 (代码文件 ConditionalXAML/MainPage.xaml.cs)：

```
if (ApiInformation.IsApiContractPresent(
  "Windows.Foundation.UniversalApiContract", 5))
{
  text1.Text = "contract 5 present";
}
```

在 XAML 代码中，不能多次设置 Text 属性，而必须确保该属性只为特定的版本设置了一次。否则，将得到 Text 属性定义多次的错误。这就是为什么定义了可以用来设置别名名称空间的逆方法。

可以根据可用的协定，设置多个名称空间(XAML 文件 ConditionalXAML/MainPage.xaml)：

```
xmlns:contract5=
"http://schemas.microsoft.com/winfx/2006/xaml/presentation?
  IsApiContractPresent(Windows.Foundation.UniversalApiContract, 5)"
xmlns:notcontract5="http://schemas.microsoft.com/winfx/2006/xaml/presentation?
  IsApiContractNotPresent(Windows.Foundation.UniversalApiContract, 5)"
```

现在使用不同的名称空间别名来设置 TextBlock 的 Text 属性。使用这些别名定义，可以确保只定义其中一个 (XAML 文件 ConditionalXAML/MainPage.xaml)：

```
<TextBlock
  x:Name="text1"
  contract5:Text="contract 5 present"
  notcontract5:Text="contract 5 not present" />
```

在哪里可以找到 API 协定的名称？每个 Windows Runtime API 都记录在 https://docs.microsoft.com/uwp/api 中，引入 API 时，文档列出了 API 协定与版本号。Windows.Foundation.UniversalApiContract 本身就是一个引用其他协定的参考协定。在文件夹%ProgramFiles(x86)%\Windows Kits\10\References 中可以找到所有协定。使用 ildasm 工具可以读取包含协定信息的.md 文件。

## 33.3　控件

由于 Windows 应用程序有很多控件可用，因此最好了解控件的层次结构和一些特定基类的 UI 类。了解这些会更容易使用 UWP 控件，知道这些类型能做什么工作。

下面讨论用于 Windows 应用程序的 UI 类的层次结构。

- DependencyObject——这个类位于 Windows Runtime XAML 元素的层次结构顶部。派生自 DependencyObject 的每个类都可以有依赖属性。在本章的 XAML 介绍中，已经介绍了依赖属性。
- UIElement——这是带有视觉外观的元素的基类。这个类提供了用户交互的功能，比如指针事件(PointerPressed、PointerMoved 等)，键处理事件(KeyDown、KeyUp)，焦点事件(GotFocus、LostFocus)，指针捕获(CapturePointer、PointerCanceled 等)，拖放(DragOver、Drop 等)。这个类还提供了新的 Lights 属性，从构建号 15063 开始就可以通过 light 高亮显示。
- FrameworkElement——类 FrameworkElement 派生自 UIElement，添加了更多的特性。从 FrameworkElement 派生的类可以参与布局系统。属性 MinWidth、MinHeight、Height 和 Width 由 FrameworkElement 类定义。FrameworkElement 也定义了生命周期事件：Loaded、SizeChanged、Unloaded 都是这些事件中的一部分。数据绑定特性是 FrameworkElement 类定义的另一组功能。这个类定义了 DataContext、DataContextChanged、SetBinding 和 GetBindingExpression API。
- Control——Control 类派生自 FrameworkElement，是 UI 控件的基类，例如 TextBox、Hub、DatePicker、SearchBox、UserControl 等。控件通常有一个默认样式，其 ControlTemplate 分配给 Template 属性。Control 类为基类 UIElement 定义的事件定义了可重写的 On*方法。控件定义了 TabIndex；用于前景、背景和边界(Foreground、Background、BorderBrush、BorderThickness)的属性；启用它并使用键盘上的 Tab 键来访问它的属性(IsTabStop、TabIndex)。
- ContentControl——类 ContentControl 派生自 Control，允许将任何内容作为该控件的子内容。ContentControl 的例子有 AppBar、Frame、ButtonBase、GroupItem 和 ToolTip 控件。ContentControl 定义了可以分配任何内容的 Content 属性、分配 DataTemplate 的 ContentTemplate 属性、动态分配数据模板的 ContentTemplateSelector 以及用于简单动画的 ContentTransitions 属性。
- ItemsControl——ContentControl 只能有一个内容，而 ItemsControl 可以查看内容列表。ContentControl 定义了要列出其子项的 Content 属性，而 ItemsControl 用 Items 属性实现了这个功能。ContentControl 和

ItemsControl 都派生自基类 Control。ItemsControl 可以显示固定数量的项或通过列表绑定的项。派生自 ItemsControl 的控件有 ListView、GridView、ListBox、Pivot 和 Selector。

- Panel——另一个可以作为项容器的类是 Panel 类。这个类派生自基类 FrameworkElement。Panel 用于定位和排列子对象。从 Panel 派生的类的例子有 Canvas、Grid、StackPanel、VariableSizedWrapGrid、VirtualizingPanel、ItemsStackPanel、ItemsWrapGrid 以及 RelativePanel。
- RangeBase——这个类派生自 Control 类，是 ProgressBar、ScrollBar 和 Slider 的基类。RangeBase 定义了当前值的 Value 属性、Minimum 和 Maximum 属性，以及 ValueChanged 事件处理程序。
- FlyoutBase——这个类直接派生自 DependencyObject，允许在其他元素上显示用户界面——换句话说，它们是随时可弹出的。

**注意：**
控件模板详见第 35 章。

在浏览了主要类别和类型的层次结构之后，下面了解细节。

### 33.3.1 框架派生的 UI 元素

有些元素不是真正的控件，但它们仍然是派生自 FrameworkElement 的类的 UI 元素。这些类不允许通过指定模板来定制外观。UI 元素的类别使用这个基类：呈现器、媒体元素和文本显示元素，如表 33-3 所示。

表 33-3

| 类 | 说 明 |
|---|---|
| Border<br>Viewbox<br>ContentPresenter<br>ItemsPresenter | 呈现器不是交互式的类，但是他们仍然提供视觉外观。<br>Border 类定义了围绕单个控件的边框(可以是包含多个其他控件的 Grid)。<br>Viewbox 能够拉伸和缩放子元素。<br>ContentPresenter 在 ControlTemplate 中使用。它定义控件的内容将显示在何处。<br>ItemsPresenter 用于确定项在 ItemsControl 中的位置。第 35 章讨论了控件和项模板 |
| TextBlock<br>RichTextBlock | TextBlock 和 RichTextBlock 控件用于显示文本。使用这些控件不能输入文本；它们只是用来展示的。TextBlock 控件不仅允许分配简单的文本，还允许分配更复杂的文本元素，如段落和内联元素。RichTextBlock 也支持溢出。注意，RichTextBlock 不支持使用 RTF(富文本文件)。此时需要使用 RichEditBox。使用这些控件显示流文本，如第 36 章所示 |
| Ellipse<br>Polygon<br>Polyline<br>Path<br>Rectangle | Shape 类派生自 FrameworkElement。Shape 本身是 Ellipse、Polygon、Polyline、Path、Rectangle 等的基类。这些类用于将向量绘制到屏幕上。这些类参见第 35 章 |
| Panel | Panel 类派生自 FrameworkElement。Panel 用于组织屏幕上的 UI 元素。派生自 Panel 类的不同面板将在本章的“布局面板”一节中讨论 |
| CaptureElement | CaptureElement 类用于呈现捕获设备(如摄像机或网络摄像机)的流。该类在第 36 章中使用 |
| InkCanvas | InkCanvas 控件是用钢笔和墨水绘制的绘图区域。阅读第 36 章了解更多关于使用墨水的信息 |
| Image | Image 控件用于显示图像。此控件支持显示如下格式的图像：JPEG、PNG、BMP、GIF、TIFF、JPEG XR、ICO 以及 SVG。自 Windows 1703 版本开始就支持 SVG(Scalable Vector Graphics，可缩放矢量图形)，从 Windows 1607 版本开始就支持 GIF 动画 |
| ParallaxView | ParallaxView 是构建版本 16299 中的一个新控件，可在滚动时产生视差效果 |

### 1. 呈现器

在 PresentersPage 中使用一些呈现器控件：Border 和 Viewbox。Border 用于对两个 TextBox 元素分组。因为 Border 元素只能包含一个子元素，所以在 Border 元素中使用 StackPanel。Border 指定了 Background、BorderBrush 和 BorderThickness。

在下面的代码片段中，两个 Viewbox 控件用于拉伸 Button 控件。第一个 Viewbox 使用 Fill 模式的延伸，将 Button 完全填充在 Viewbox 中，而第二个 Viewbox 使用 Uniform 模式的延伸。对于 Uniform，要保持纵横比(代码文件 ControlsSamples /Views/PresentersPage.xaml)：

```
<Border Background="LightSeaGreen" BorderBrush="DarkGreen" BorderThickness="12"
  Margin="12" Padding="8">
  <StackPanel Orientation="Vertical">
    <TextBox Header="Title" x:Name="Title" FontSize="34" />
    <TextBox Header="Publisher" x:Name="Publisher" FontSize="34" />
  </StackPanel>
</Border>
<Viewbox Grid.Row="1" Stretch="Fill" StretchDirection="Both">
  <Button Margin="4" FontSize="14">Button with fill stretch</Button>
</Viewbox>
<Viewbox Grid.Row="2" Stretch="Uniform" StretchDirection="Both">
  <Button Margin="4" FontSize="14">Button with uniform stretch</Button>
</Viewbox>
```

图 33-7 显示了正在运行的应用程序的呈现器页面。在这里可以看到 TextBox 控件是如何被包围的，按钮显示在两个不同的 Viewbox 配置中。

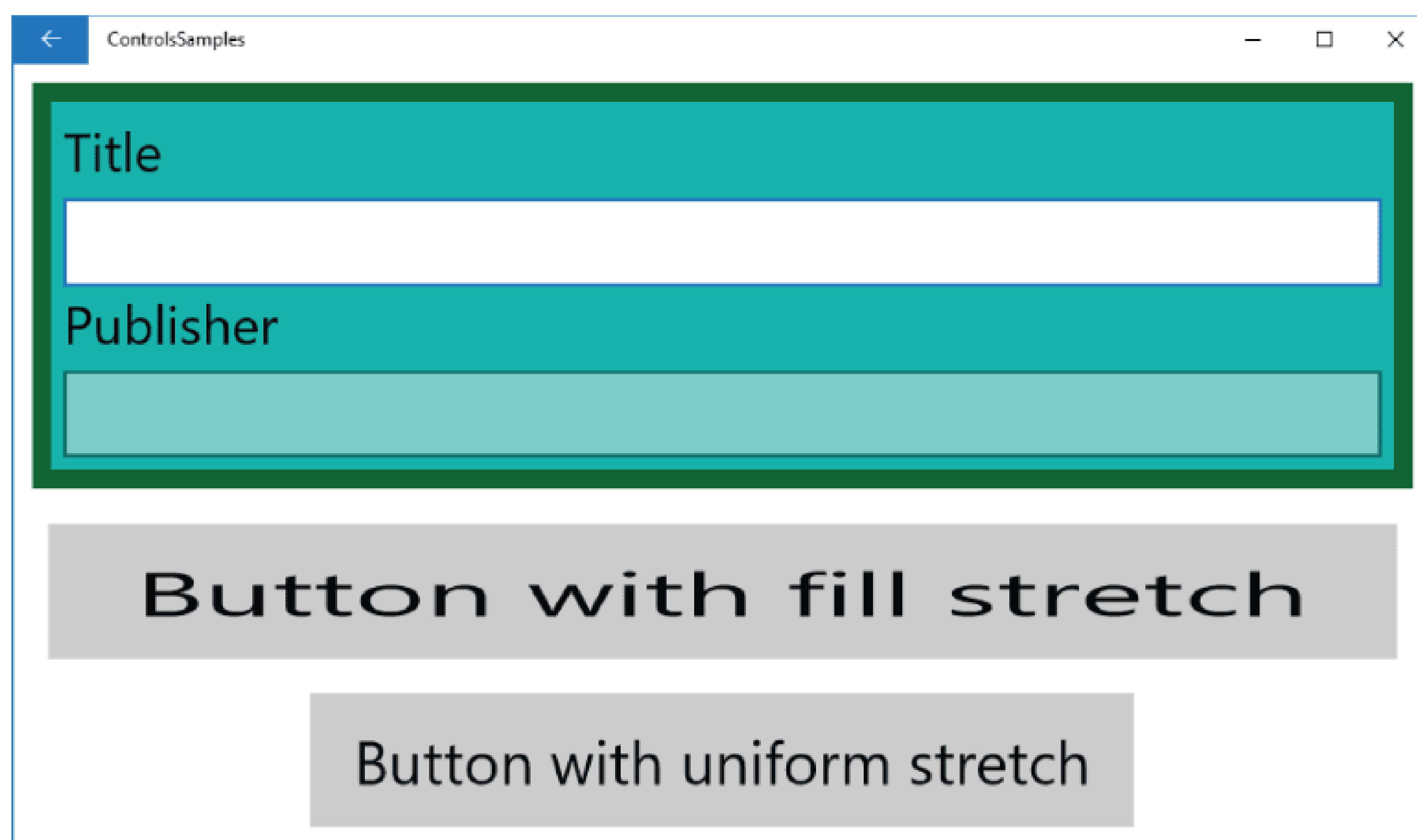



图 33-7

> **注意：**
> 控件派生的类有一个隐式边界，可以使用 BorderThickness 和 BorderBrush 属性来定制它。

### 2. 视差

ParallaxView 提供了视差效应。有了视差效应，就可以在前景中显示一个列表，在背景中显示一幅图片。背景中的图片比前景移动得慢，以获得立体效果，这称为视差效应。

示例应用程序 ParallaxViewSample 演示了这种效果，使用滑块可以增加或减少该效果。

示例应用程序中显示的是 LunchMenu 对象的一个简单模型(代码文件 ParallaxViewSample/Models/LunchMenu.cs)：

```
public class LunchMenu
{
  public int MenuId { get; set; }
  public string Text { get; set; }
  public string ImageUrl { get; set; }
}
```

ParallaxView 控件将 Image 定义为子元素，也可以为背景创建其他形状。ParallaxView 的 Source 属性绑定到 ListView 控件。ListView 控件显示项的列表，并绑定到 LunchMenu 对象列表上。要使用 ParallaxView 定义该效果，需要设置 HorizontalShift 和 VerticalShift 属性。如源代码所示，ListView 垂直滚动，因此下面讨论垂直移动。当 VerticalShift 设置为 0 时，没有视差效应。将该值设置为 100 意味着，ParallaxView 配置为比 ListView 小 100 像素，因此 ListView 的滚动速度更快。下面的代码片段将 HorizontalShift 和 VerticalShift 属性绑定到一个 Slider，因此可以在应用程序运行时配置该效果(代码文件 ParallaxViewSample/MainPage.xaml)：

```
<StackPanel Orientation="Vertical">
  <Slider x:Name="HorizontalSlider" Header="Horizontal Shift" Minimum="0"
    Maximum="1000" Value="0" />
  <Slider x:Name="VerticalSlider" Header="Vertical Shift" Minimum="0"
    Maximum="1000" Value="0" />
</StackPanel>

<ParallaxView Source="{x:Bind MenuItems}" Grid.Row="1"
              HorizontalShift="{x:Bind HorizontalSlider.Value, Mode=OneWay}"
              VerticalShift="{x:Bind VerticalSlider.Value, Mode=OneWay}">
  <Image Source="https://kantinem101.blob.core.windows.net/menuimages/" +
    "Hirschragout_1024">
  </Image>
</ParallaxView>

<ListView HorizontalAlignment="Center" x:Name="MenuItems"
  ItemsSource="{x:Bind Menus, Mode=OneWay}" SelectionMode="None"  Grid.Row="1">
  <ListView.ItemTemplate>
    <DataTemplate x:DataType="model:LunchMenu">
      <Grid Margin="12">
        <Image Width="300" Source="{x:Bind ImageUrl, Mode=OneWay}" />
        <TextBlock Margin="8" VerticalAlignment="Bottom"
          HorizontalTextAlignment="Center" Text="{x:Bind Text, Mode=OneWay}"
          Style="{StaticResource SubtitleTextBlockStyle}" FontWeight="Bold" />
      </Grid>
    </DataTemplate>
  </ListView.ItemTemplate>
</ListView>
```

**注意：**
本章的“数据绑定”一节详细解释数据绑定。

运行该应用程序时，可以看到背景图像和前景以不同的速度移动，如图 33-8 所示。

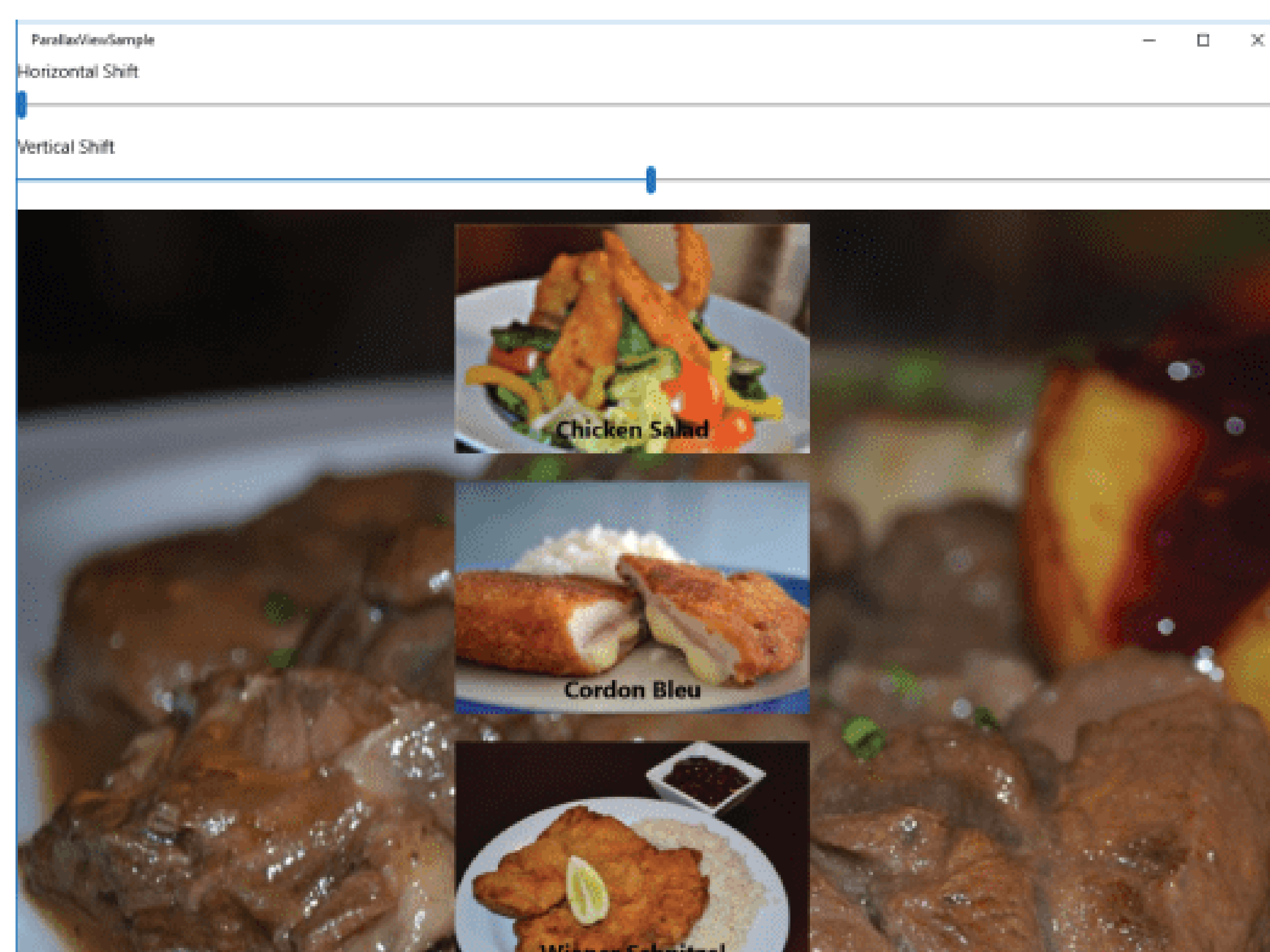



图 33-8

要在水平模式下查看视差效果，可以将 ListView 更改为水平滚动。为此，可以更改 ListView 的 ItemsPanel，如下面的代码片段所示。在可下载的代码示例中，需要取消对这个 ListView 配置的注释，使它成为活动的(代

码文件 ParallaxViewSample/MainPage.xaml)：

```
<ListView HorizontalAlignment="Center" x:Name="MenuItems"
  ItemsSource="{x:Bind Menus, Mode=OneWay}" SelectionMode="None"
  ScrollViewer.VerticalScrollBarVisibility="Disabled"
  ScrollViewer.VerticalScrollMode="Disabled" Grid.Row="1"
  ScrollViewer.HorizontalScrollBarVisibility="auto"
  ScrollViewer.HorizontalScrollMode="Enabled">
  <ListView.ItemsPanel>
    <ItemsPanelTemplate>
      <ItemsStackPanel Orientation="Horizontal" />
    </ItemsPanelTemplate>
  </ListView.ItemsPanel>
  <ListView.ItemTemplate>
    <DataTemplate x:DataType="model:LunchMenu">
      <Grid Margin="12">
        <Image Width="300" Source="{x:Bind ImageUrl, Mode=OneWay}" />
        <TextBlock Margin="8" VerticalAlignment="Bottom"
          HorizontalTextAlignment="Center" Text="{x:Bind Text, Mode=OneWay}"
          Style="{StaticResource SubtitleTextBlockStyle}" FontWeight="Bold" />
      </Grid>
    </DataTemplate>
  </ListView.ItemTemplate>
</ListView>
```

图 33-9 显示了 ParallaxViewSample 的水平版本。

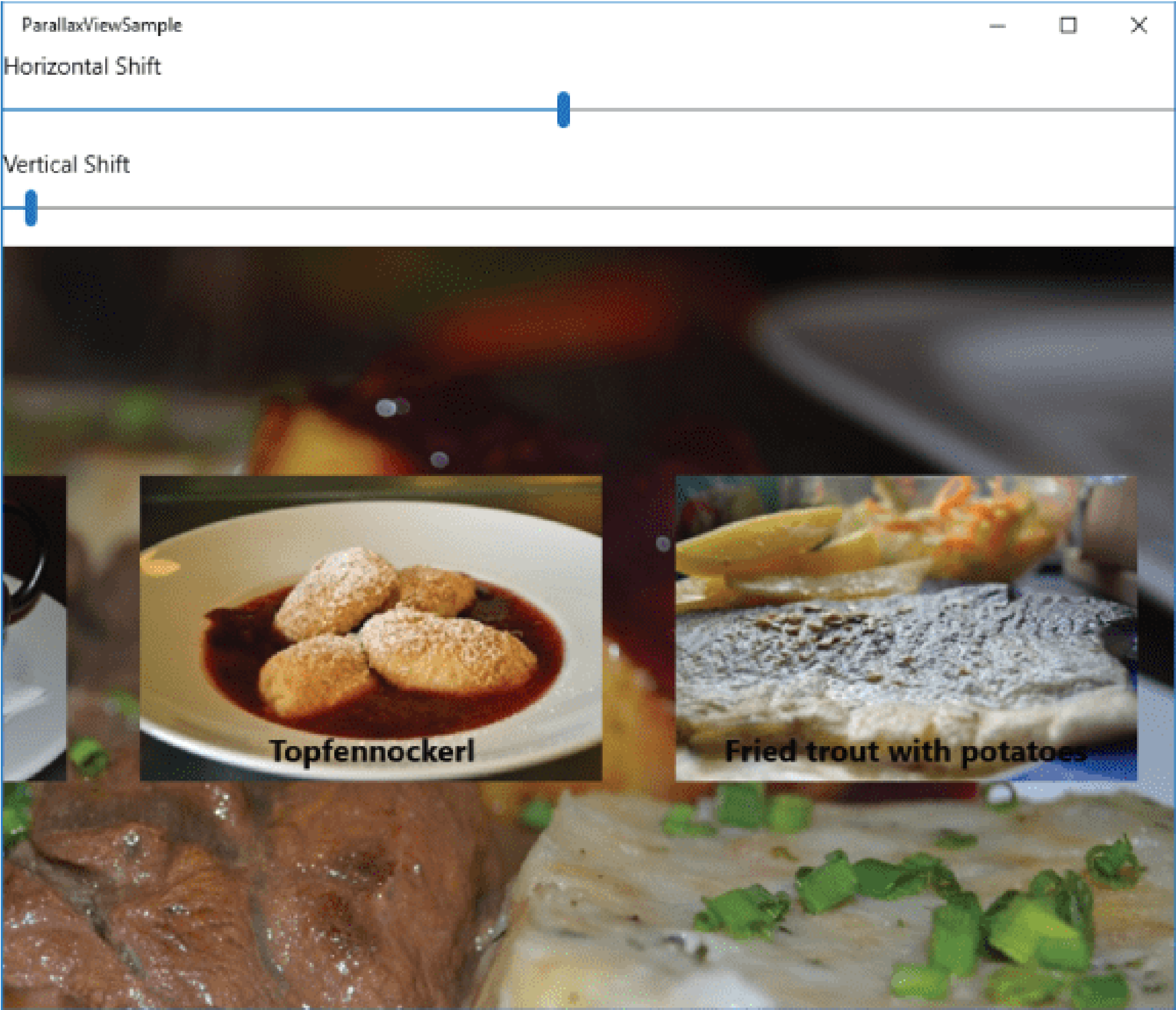



图 33-9

## 33.3.2　控件派生的控件

直接从基类 Control 派生的控件属于这个类别。表 33-4 描述了其中的一些控件。

表 33-4

| 控　件 | 说　明 |
|---|---|
| TextBox | 此控件用于显示简单的、未格式化的文本。此控件可用于用户输入。Text 属性包含用户输入。PlaceholderText 允许向用户提供要在输入字段中输入的信息。通常输入文本的一些信息会显示在附近。这可以直接使用 Header 属性完成 |
| RichEditBox | 与 TextBox 控件相反，RichEditBox 允许输入格式化的文本、超链接和图像。文本对象模型(Text Object Model ，TOM)是在 Document 属性中使用的。第 36 章详细介绍了如何使用这个控件。可以使用 Microsoft Word 创建能读入 RichEditBox 的 RTF 文件 |

(续表)

| 控 件 | 说 明 |
| --- | --- |
| PasswordBox | 此控件用于输入密码。它具有密码输入的特定属性，比如 PasswordChar 定义在用户输入密码时显示的字符。可以使用 Password 属性检索输入的密码。此控件还具有与 TextBox 控件类似的 Header 和 PlaceholderText 属性 |
| ProgressRing | 此控件指示操作正在进行。它显示为一个环形的“旋转器”。另一个显示正在进行的操作的控件是 ProgressBar，但是这个控件属于范围控件 |
| DatePicker<br>CalendarDatePicker<br>CalendarView | DatePicker 和 CalendarDatePicker 控件允许用户选择日期。如果用户知道日期，则显示日历是没有用的，就可以使用 DatePicker 选择日期。 CalendarDatePicker 在内部使用 CalendarView。如果日历应该一直可见，或者需要选择多个日期，就可以使用 CalendarView。请注意，还有一个 DatePickerFlyout(派生自 Flyout 的控件)，它允许用户在新打开的窗口中选择日期 |
| TimePicker | TimePicker 允许用户输入时间。类似于 DatePicker，通过 TimePicker 还可以使用 TimePickerFlyout |
| AppBarSeparator | AppBarSeparator 控件可以用作 CommandBar 中的分隔符 |
| ColorPicker | ColorPicker 允许用户选择颜色 |
| Hub<br>HubSection | Hub 控件允许在平移视图中对内容进行分组。该控件中的内容是在多个 HubSection 控件中定义的。Hub 控件与许多应用程序一起使用，以“Hero”图像布局应用程序的主视图 |
| UserControl | UserControl 是可以用于重用的控件，并使用页面简化 XAML 代码。用户控件可以添加到页面中，本章和下一章使用用户控件 |
| Page | Page 类本身派生自 UserControl，因此它也是 UserControl。Page 用于在 Frame 中导航 |
| PersonPicture | PersonPicture 是 16929 构建版本中的新控件。它用来显示一个人的头像。此控件与 ContactManager 和 Contact API 一起使用 |
| RatingControl | RatingControl 是 16929 构建版本中的另一个新控件。使用此控件，用户可以输入星级评分 |
| SemanticZoom | SemanticZoom 控件定义了两个视图：缩小视图和放大视图。这允许用户快速导航到大型数据集，例如，在缩小视图中只显示第一个字符。在放大视图中，用户用选定的字母定位数据对象 |
| SplitView | SplitView 控件有一个窗格和一个内容。窗格可以打开和关闭。当打开窗格时，内容可以部分位于窗格后面，也可以向右移动。打开的窗格可以是小的(紧凑的)或宽的。SplitView 在新的 NavigationPane 中使用 |

### 1. 使用文本框

包含 Control 派生控件的第一个示例显示了几个 TextBox 控件。在 TextBox 类中，可以将 InputScope 属性指定为大量值列表中的值，如 EmailNameOrAddress、CurrencyAmountAndSymbol 或 Formula.。如果应用程序在平板模式下使用，并带有屏幕键盘，键盘会根据输入字段的需要调整不同的布局并显示键。示例代码中的最后一个文本框是多行 TextBox。为了让用户按下回车键，可以设置 AcceptsReturn 属性。同时，如果文本在一行中放不下，就设置 TextWrapping 属性，使文本换行。文本框的高度设置为 150。如果输入的文本在这个文本框中放不下，则使用附加属性 ScrollViewer.VerticalScrollBarVisibility 来显示滚动条 (代码文件 ControlsSamples/Views/TextPage.xaml)：

```
<TextBox Header="Email" InputScope="EmailNameOrAddress"></TextBox>
<TextBox Header="Currency" InputScope="CurrencyAmountAndSymbol"></TextBox>
<TextBox Header="Alpha Numeric" InputScope="AlphanumericFullWidth"></TextBox>
<TextBox Header="Formula" InputScope="Formula"></TextBox>
<TextBox Header="Month" InputScope="DateMonthNumber"></TextBox>
<TextBox Header="Multiline" AcceptsReturn="True" TextWrapping="Wrap"
  Height="150" ScrollViewer.VerticalScrollBarVisibility="Auto"  />
```

如图 33-10 所示为多行文本框的结果，其中包含多行和一个滚动条。

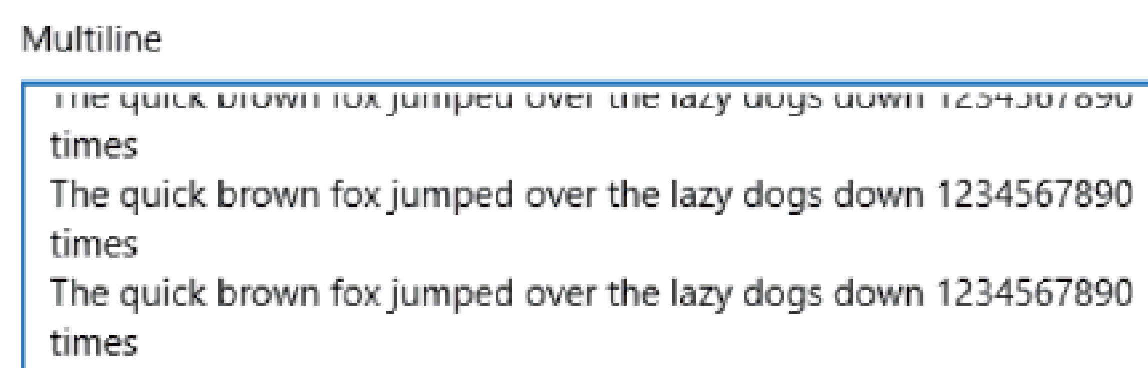



图 33-10

### 2. 选择日期

对于选择日期，可以使用多个选项。下面看看不同的选项，以及 CalendarView 控件的特殊特性。

在以下代码片段中，CalendarView 配置为允许选择多个日期。每周的第一个工作日设置为周一，最小的一天设置为绑定属性 MinDate，事件 CalendarViewDayItemChanging 和 SelectedDatesChanged 分配给事件处理程序(代码文件 DateSelectionSample/MainPage.xaml)：

```
<CalendarView x:Name="CalendarView1" Margin="12" HorizontalAlignment="Center"
  SelectionMode="Multiple"
  FirstDayOfWeek="Monday"
  MinDate="{x:Bind MinDate, Mode=OneTime}"
  CalendarViewDayItemChanging="OnDayItemChanging"
  SelectedDatesChanged="OnDatesChanged" />
```

在代码隐藏文件中，MinDate 属性设置为一个预定义的日期。用户不能使用日历提前一天到达(代码文件 DateSelectionSample/MainPage.xaml.cs)：

```
public DateTimeOffset MinDate { get; } =
  DateTimeOffset.Parse("1/1/1965, new CultureInfo("en-US"));
```

在 OnDayItemChanging 事件处理程序中，应该将某些日期标记为 special。当天之前的日期应该排除在选择之外，根据实际的预订情况，当天应该用彩线标记。

为了获得预订，将定义 GetBookings 方法，以返回示例数据。在真正的应用程序中，可以从 Web API 或数据库中获得数据。GetBookings 方法只返回从现在开始若干天(2,3,5...)的预订，通过返回一个元组得到一天内的预订数量(1,4,3...) (代码文件 DateSelectionSample/MainPage.xaml.cs)：

```
private IEnumerable<(DateTimeOffset day, int bookings)> GetBookings()
{
  int[] bookingDays = { 2, 3, 5, 8, 12, 13, 18, 21, 23, 27 };
  int[] bookingsPerDay = { 1, 4, 3, 6, 4, 5, 1, 3, 1, 1 };

  for (int i = 0; i < 10; i++)
  {
    yield return (DateTimeOffset.Now.Date.AddDays(bookingDays[i]),
      bookingsPerDay[i]);
  }
}
```

**注意：**
元组和本地函数参见第 13 章。yield 语句参见第 12 章。

当显示 CalendarView 的项时，将调用 OnDayItemChanging 方法。每个显示的日期都调用此方法。方法 OnDayItemChanging 是使用本地函数实现的。该方法的主块包含一个 switch 语句，基于数据绑定阶段来进行切换。CalendarView 控件支持多个阶段，允许在不同的迭代中调整用户界面。第一阶段很快；在此阶段之后，已经可以向用户显示一些信息。接下来的每个阶段都是如此。在以后的阶段中，可以从 Web API 中检索信息，并更新这些信息。

在 OnDayItemChanging 的实现中，第一个阶段调用本地函数 RegisterUpdateCallback 来注册对 OnDayItemChanging 事件处理程序的下一个调用。在第二阶段，使用本地函数 SetBlackoutDates 将日期涂黑。第三个阶段检索预订(代码文件 DateSelectionSample/MainPage.xaml.cs)：

```
private void OnDayItemChanging(CalendarView sender,
  CalendarViewDayItemChangingEventArgs args)
{
  switch (args.Phase)
```

```
    {
      case 0:
        RegisterUpdateCallback();
        break;
      case 1:
        SetBlackoutDates();
        break;
      case 2:
        SetBookings();
        break;
      default:
        break;
    }

    // local functions...
  }
```

本地函数 RegisterUpdateCallback 只是调用 CalendarViewDayItemChangingEventArgs 参数的 RegisterUpdate Callback，传递事件处理程序方法，因此再次调用此方法(代码文件 DateSelectionSample/MainPage.xam.cs)：

```
private void OnDayItemChanging(CalendarView sender,
  CalendarViewDayItemChangingEventArgs args)
{
  //...
  void RegisterUpdateCallback() =>
    args.RegisterUpdateCallback(OnDayItemChanging);
  //...
}
```

本地函数 SetBlackoutDates 涂黑今天之前的日期，以及所有的星期六和星期天。从 args.Item 属性返回的 CalendarViewDayItem 定义了 IsBlackout 属性(代码文件 DateSelectionSample/MainPage.xam.cs)：

```
private void OnDayItemChanging(CalendarView sender,
  CalendarViewDayItemChangingEventArgs args)
{
  //...
  void SetBlackoutDates()
  {
    if (args.Item.Date < DateTimeOffset.Now || args.Item.Date.DayOfWeek ==
      DayOfWeek.Saturday || args.Item.Date.DayOfWeek == DayOfWeek.Sunday)
    {
      args.Item.IsBlackout = true;
    }
    RegisterUpdateCallback();
  }
  //...
}
```

最后，SetBookings 方法检索关于预订的信息。如果在预订中也发现了接收日期，则会检查在 CalendarViewDayItem 中找到的接收日期。如果是，则调用 SetDensityColors，把红色或绿色的列表(取决于工作日)添加到日期项中。最后，再次调用 RegisterUpdateCallback 本地函数；否则，只会在第三阶段调用该函数，显示第一天(代码文件 DateSelectionSample/MainPage.xaml.cs)：

```
private void OnDayItemChanging(CalendarView sender,
  CalendarViewDayItemChangingEventArgs args)
{
  //...

  void SetBookings()
  {
    var bookings = GetBookings().ToList();

    var booking = bookings.SingleOrDefault(
      b => b.day.Date == args.Item.Date.Date);
    if (booking.bookings > 0)
    {
      var colors = new List<Color>();
      for (int i = 0; i < booking.bookings; i++)
      {
        if (args.Item.Date.DayOfWeek == DayOfWeek.Saturday ||
          args.Item.Date.DayOfWeek == DayOfWeek.Sunday)
        {
          colors.Add(Colors.Red);
        }
        else
```

```
          {
            colors.Add(Colors.Green);
          }
        }

        args.Item.SetDensityColors(colors);
      }
      RegisterUpdateCallback();
    }
  }
```

当用户选择日期时，将调用 OnDatesChanged 方法。在这个方法中，所有选中的日期都将在 CalendarViewSelectedDatesChangedEventArgs 中接收。选中的日期写入 currentDatesSelected 列表，取消选择的日期将再次从列表中删除。使用 string.Join，所有选中的日期都显示在 MessageDialog 中(代码文件 DateSelectionSample/MainPage.xaml.cs)：

```
private List<DateTimeOffset> currentDatesSelected = new List<DateTimeOffset>();

private async void OnDatesChanged(CalendarView sender,
  CalendarViewSelectedDatesChangedEventArgs args)
{
  currentDatesSelected.AddRange(args.AddedDates);
  args.RemovedDates.ToList().ForEach(date =>
    currentDatesSelected.Remove(date));

  string selectedDates = string.Join(", ",
    currentDatesSelected.Select(d => d.ToString("d")));

  await new MessageDialog($"dates selected: {selectedDates}").ShowAsync();
}
```

运行这个应用程序时，可以看到日历，如图 33-11 所示，前几天以及周六/周日都被涂黑了，而预订的信息用彩线显示。

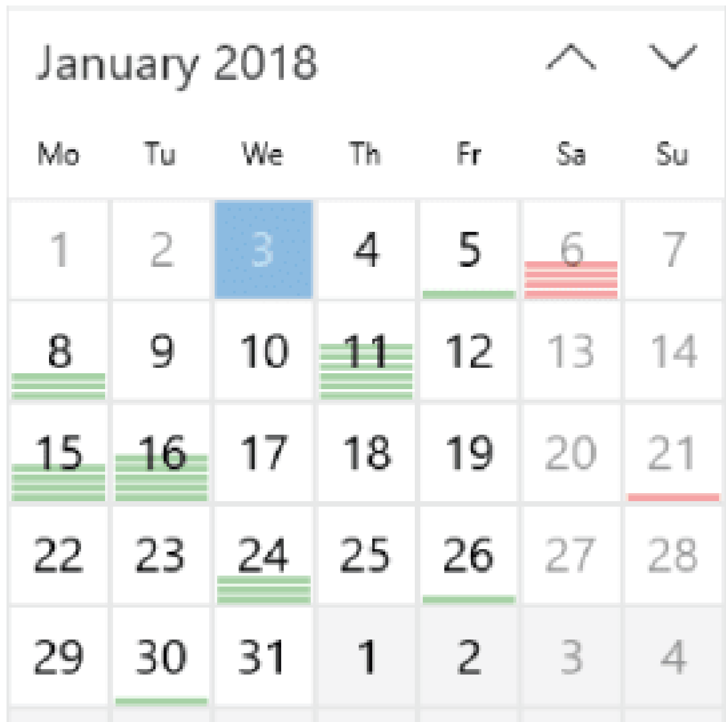



图 33-11

单击日历的月份时，将显示完整的年份，如图 33-12 所示。单击顶部的年份时，可以看到一个纪元(参见图 33-13)。所以很容易选择很远的日期。

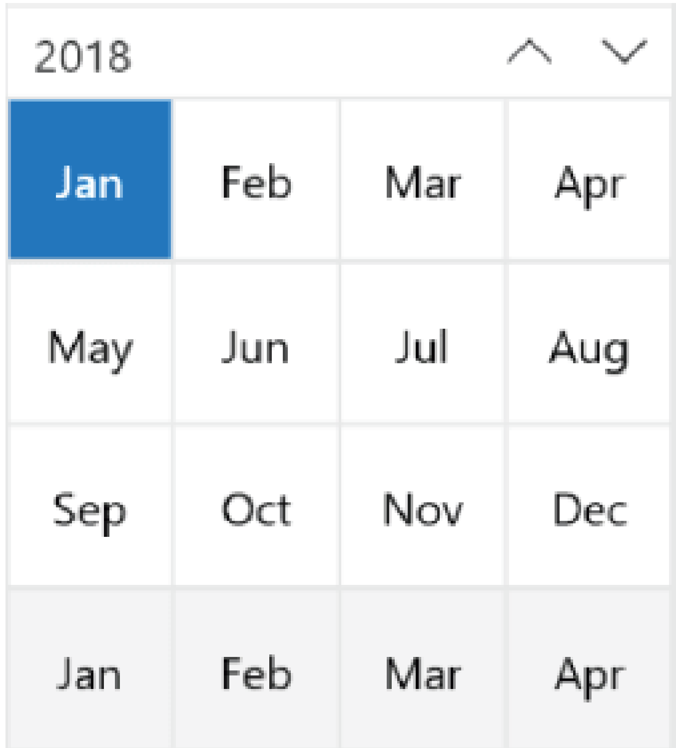



图 33-12

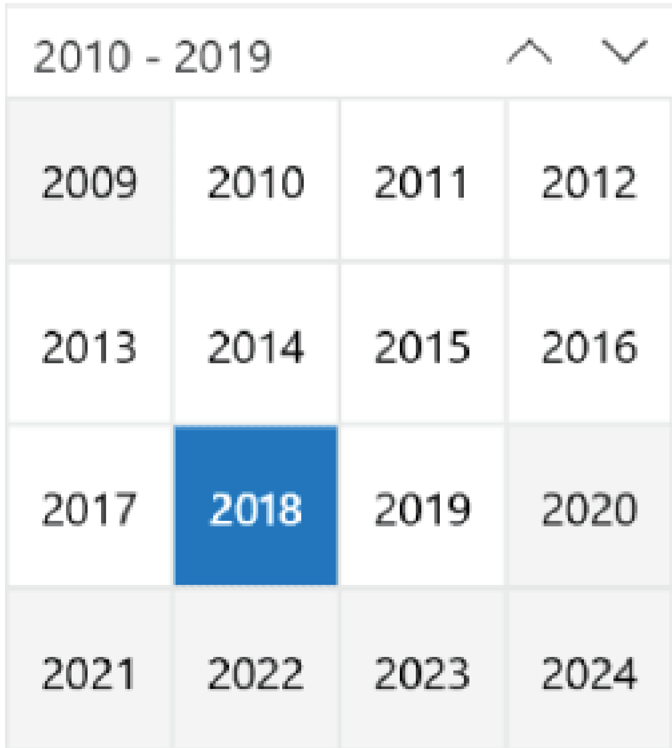



图 33-13

当使用 CalendarDatePicker 时，没有像 CalendarView 那么多特性，但是它不会占用屏幕的空间，除非用户打开它来选择日期。CalendarDatePicker 定义了 DateChanged 事件；只能选择一个日期(代码文件 DateSelectionSample/MainPage.xaml)：

```
<CalendarDatePicker x:Name="CalendarDatePicker1" Grid.Row="0" Grid.Column="1"
  DateChanged="OnDateChanged" Margin="12" />
```

在 OnDateChanged 事件处理程序中，会接收到 CalendarDatePickerDateChangedEventArgs，它包含 NewDate 属性(代码文件 DateSelectionSample/MainPage.xaml.cs)：

```
private async void OnDateChanged(CalendarDatePicker sender,
  CalendarDatePickerDateChangedEventArgs args)
{
  await new MessageDialog($"date changed to {args.NewDate}").ShowAsync();
}
```

图 33-14 显示了打开日历时的用户界面。

DatePicker 的 XAML 代码非常相似。它只是不显示日历来选择日期，而是有一个完全不同的视图(代码文件 DateSelectionSample/MainPage.xaml)：

```
<DatePicker DateChanged="OnDateChanged1" x:Name="DatePicker1" Grid.Row="1"
  Margin="12" />
```

DatePicker 的事件处理程序接收对象和 DatePickerValueChangedEventArgs 参数(代码文件 DateSelectionSample/MainPage.xaml)：

```
private async void OnDateChanged1(object sender,
  DatePickerValueChangedEventArgs e)
{
  await new MessageDialog($"date changed to {e.NewDate}").ShowAsync();
}
```

图 33-15 显示了打开时的 DatePicker。如果用户不查看日历(例如生日)就知道日期，那么滚动年份、月份和日期要快得多。

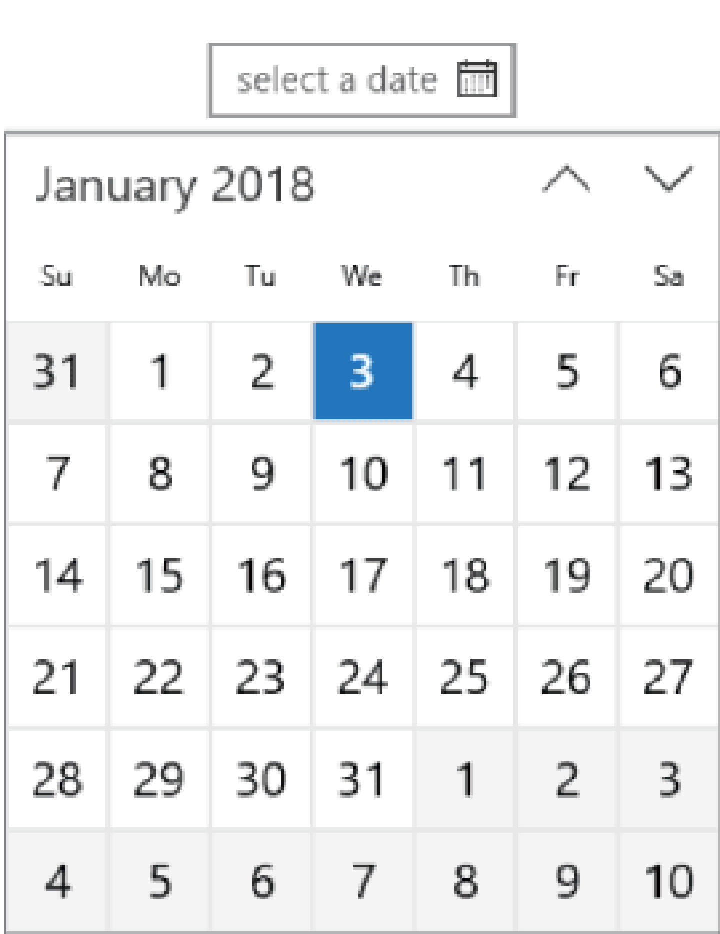



图 33-14




图 33-15

选择日期的最后一个选项是 Flyout。Flyout 可以与其他控件一起使用。这里使用一个按钮控件，按钮的 Flyout 属性定义为使用 DatePickerFlyout。

```
<Button Content="Select a Date" Grid.Row="1" Grid.Column="1" Margin="12">
  <Button.Flyout>
    <DatePickerFlyout x:Name="DatePickerFlyout1" DatePicked="OnDatePicked" />
  </Button.Flyout>
</Button>
```

单击按钮将打开弹出窗口，如图 33-16 所示。

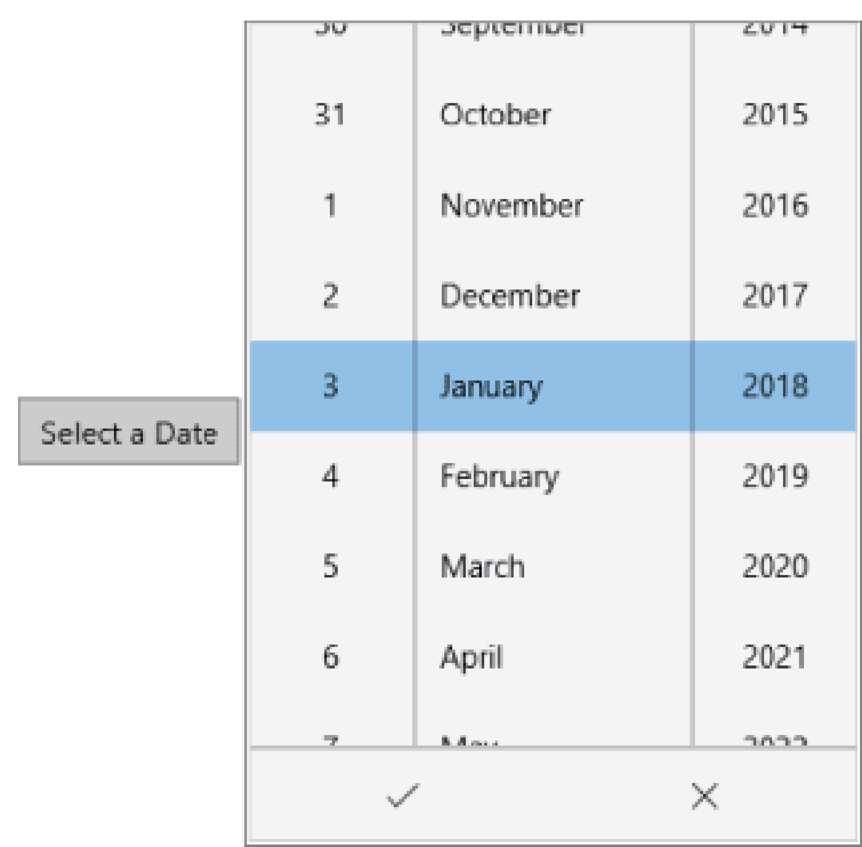



图 33-16

### 33.3.3　范围控件

范围控件(见表 33-5)，如 ScrollBar、ProgressBar 和 Slider 都派生自同一个基类 RangeBase。

表 33-5

| 控　　件 | 说　　明 |
|---|---|
| ScrollBar | ScrollBar 控件包含一个 Thumb，用户可以从 Thumb 中选择一个值。例如，如果文档在屏幕中放不下，就可以使用滚动条。一些控件包含滚动条，如果内容过多，就显示滚动条 |
| ProgressBar | 使用 ProgressBar 控件，可以指示时间较长的操作的进度 |
| Slider | 使用 Slider 控件，用户可以移动 Thumb，选择一个范围的值 |

#### 1. ProgressBar

示例应用程序显示了两个 ProgressBar 控件。将第二个控件的 IsIndeterminate 属性设置为 true。如果不知道一个活动需要多长时间，最好使用这个属性。如果想知道操作需要多长时间，可以在 ProgressBar 中设置当前状态值，而不需要设置 IsIndeterminate 模式；默认值为 False(代码文件 ControlsSamples/Views/Range Controls Page.xaml)：

```
<ProgressBar x:Name="progressBar1" Grid.Row="0" Margin="12" />
<ProgressBar IsIndeterminate="True" Grid.Row="1" Margin="12" />
```

在加载页面时，将调用 ShowProgress 方法。这里，第一个 ProgressBar 的当前值是使用 DispatcherTimer 设置的。将 DispatcherTimer 配置为每秒触发一次，ProgressBar 的 Value 属性每秒都递增(代码文件 ControlsSamples/Views/RangeControlsPage.xaml.cs)：

```
private void ShowProgress()
{
  var timer = new DispatcherTimer();
  timer.Interval = TimeSpan.FromSeconds(1);
  int i = 0;
  timer.Tick += (sender, e) =>
  {
    progressBar1.Value = i++;
    if (i >= 100)
    {
      i = 0;
    }
  };
  timer.Start();
}
```

**注意：**
DispatcherTimer 类详见第 21 章。

运行应用程序时，可以看到两个 ProgressBar 控件处于活动状态。使用第一个 ProgressBar 控件可以看到状态，第二个则显示水平漂浮的点(参见图 33-17)。

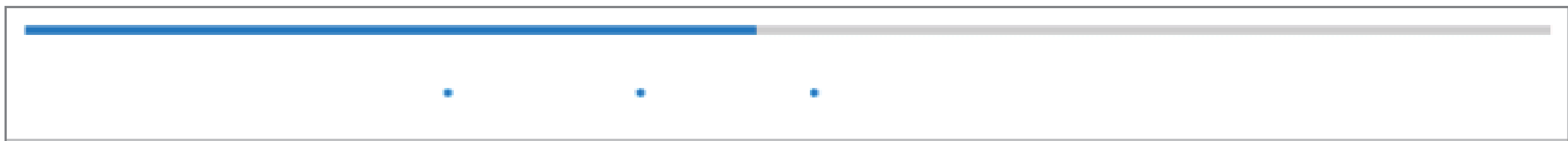


图 33-17

### 2. Slider

使用 Slider 控件，可以指定 Minimum 和 Maximum 值，并使用 Value 属性来分配当前值。代码示例使用一个文本框来显示滑块的当前值(代码文件 ControlsSamples /Views/RangeControlsPage.xaml)：

```
<Slider x:Name="slider" Minimum="10" Maximum="140" Value="60"
  Grid.Row="2" Margin="12" />
<TextBox Header="Slider Value" IsReadOnly="True"
  Text="{x:Bind slider.Value, Mode=OneWay}" Grid.Row="3" Margin="12" />
```

在图 33-18 中可以看到 Slider 和 TextBox；注意它们是如何相互关联的，因为 TextBox 显示了 Slider 的实际值。




图 33-18

## 33.3.4　内容控件

内容控件(见表 33-6)的 Content 属性允许添加任何单一内容。不允许使用多个内容对象作为 Content 属性的直接子对象，但是可以添加(例如)StackPanel，它本身可以把多个控件作为子控件。

表 33-6

| 控　件 | 说　明 |
|---|---|
| ScrollViewer | ScrollViewer 是一个内容控件，可以包含单项，并提供水平和垂直的滚动条。还可以使用带有附加属性的 ScrollViewer，如前面介绍的 ParallaxViewSample 所示 |
| Frame | Frame 控件用于页面之间的导航 |
| SelectorItem<br>ComboBoxItem<br>FlipViewItem<br>GridViewItem<br>ListBoxItem<br>ListViewItem<br>GroupItem<br>PivotItem | 这些控件是 ContentControl 对象，作为属于某 ItemsControl 的项。例如，ComboBox 控件包含 ComboBoxItem 对象，ListBox 控件包含 ListBoxItem 对象，Pivot 控件包含 PivotItem 对象。GroupItem 对象通常不直接使用；使用带有分组配置的 ItemsControl 派生控件时，会使用它们 |
| ToolTip | 当用户悬停在信息上，显示工具提示时，ToolTip 会弹出一个窗口。可以使用 ToolTipService.ToolTip 附加属性配置 ToolTip。工具提示只能是文本；这是一个内容控件 |
| CommandBar | 使用 CommandBar，可以安排 AppBarButton 控件和属于命令元素的控件(如 AppBarSeparator)。CommandBar 为这些控件提供了一些布局特性。在 Windows 8 中，使用 AppBar 而不是 CommandBar——这就是为什么按钮有这些名称的原因。CommandBar 派生自 AppBar。但是，如果 CommandBar 中的布局不能满足需求，也可以使用其他控件来布局命令 |
| ContentDialog | 使用 ContentDialog 打开一个对话框。可以使用对话框所需的任何 XAML 控件自定义此控件 |
| SwipeControl | SwipeControl 允许通过触摸交互执行上下文命令——例如，在用户向左或向右滑动时为某些项打开特定的操作 |

> **注意：**
> 下一节包含一个示例，它使用按钮填充内容控件的内容——按钮本身就是一个内容控件。

## 33.3.5　按钮

按钮组成了一个层次结构。ButtonBase 类派生自 ContentControl，因此按钮有一个 Content 属性，可以包含任何单个内容。ButtonBase 类还定义了 Command 属性；因此，所有按钮都可以有一个相关的命令。下面比较一下不同的按钮，见表 33-7。

表 33-7

| 控　件 | 说　明 |
| --- | --- |
| Button | Button 类是最常用的按钮。这个类派生自 ButtonBase(其他按钮也一样)。ButtonBase 是所有按钮的基类 |
| HyperlinkButton | HyperlinkButton 显示为链接。可以在浏览器中打开 Web 页面、打开其他应用程序或导航到其他页面 |
| RepeatButton | RepeatButton 是一个按钮，当用户按下按钮时，Click 事件连续触发。使用常规按钮，Click 事件只触发一次 |
| AppBarButton | AppBarButton 用于激活应用程序中的命令。可以将该按钮添加到 CommandBar，并使用图标和标签来显示用户的信息 |
| AppBarToggleButton<br>CheckBox<br>RadioButton | CheckBox、RadioButton 和 AppBarToggleButton 派生自基类 ToggleButton。ToggleButton 可以使用“bool?”表示三种状态：Checked、Unchecked 和 Indeterminate。AppBarToggleButton 是 CommandBar 的切换按钮 |

### 1. 替换按钮的内容

按钮是一个内容控件，可以有任何内容。下面的示例向包含 Ellipse 和 TextBlock 的按钮添加 Grid 控件。该按钮还定义了 Click 事件，以演示它的不同外观，但它的行为是相同的(代码文件 ControlsSample/Views/ButtonsPage.xaml)：

```
<Button Margin="12" Click="OnButtonClick">
  <Grid>
    <Ellipse Width="200" Height="90" Fill="red" />
    <TextBlock HorizontalAlignment="Center" VerticalAlignment="Center"
      Text="Click Me!" FontSize="24" />
  </Grid>
</Button>
```



图 33-19

在图 33-19 中，可以看到按钮的新外观。Content 属性替换了前景，但是按钮仍然具有默认的背景。

> **注意：**
> 要替换按钮的完整外观(包括背景)，并使按钮变成非矩形的形状，需要为按钮创建一个 ControlTemplate。详见第 35 章。

### 2. 通过 HyperlinkButton 进行链接

使用 HyperlinkButton 控件，可以轻松激活其他应用程序。将 NavigateUri 属性设置为 URL，单击按钮，会打开默认浏览器，以打开 Web 页面。

```
<HyperlinkButton NavigateUri="https://csharp.christiannagel.com"
  Content="C# Infos" Grid.Column="1"
  Style="{StaticResource TextBlockButtonStyle}" FontSize="24" />
```

默认情况下，HyperlinkButton 看起来像浏览器中的一个链接。使用 HyperlinkButton 可以设置 NavigateUri 或定义 Click 事件，但不能同时执行这两个操作。作为 Click 事件的操作，可以以编程方式导航到另一个页面。

不仅可以为 NavigateUri 属性分配 http://或 https://值，还可以使用 ms-appx://激活其他应用程序。

### 33.3.6 项控件

与 ContentControl 相反，ItemsControl 控件(见表 33-8)可以包含项的列表。通过 ItemsControl，可以使用 Items 属性来确定某些项，也可以使用数据绑定和 ItemsSource 属性来确定某些项。但不能同时使用这两种方式。

表 33-8

| 控 件 | 说 明 |
| --- | --- |
| ItemsControl | ItemsControl 是所有其他项控件的基类，也可以直接用于显示项的列表 |
| Pivot | Pivot 控件是为应用程序创建类似于表的行为的控件 |
| AutoSuggestBox | AutoSuggestBox 替换了先前的 SearchBox.。使用 AutoSuggestBox，用户可以输入文本，控件提供自动完成功能。这个控件的使用详见第 36 章 |
| ListBox<br>ComboBox<br>FlipView | ListBox、ComboBox 和 FlipView 是三个派生自基类 Selector 的控件。Selector 派生自 ItemsControl，并添加 SelectedItem 和 SelectedValue 属性，以便从集合中选择某项。ListBox 显示了用户可以从中选择的列表。ComboBox 结合了一个文本框和一个下拉列表，允许选择列表，且使用更少的屏幕空间。FlipView 控件允许使用触摸交互来浏览项目列表，而只显示一项 |
| ListView<br>GridView | ListView 和 GridView 派生自基类 ListViewBase，ListViewBase 派生自 Selector。因此这些是最强大的选择器。ListViewBase 提供了附加的拖放项、重新排序项、添加页眉和页脚，并允许选择多个项。ListView 垂直显示项目(但也可以创建一个模板，水平显示列表)。GridView 用行和列显示数据项 |

### 33.3.7 Flyout 控件

Flyout 控件(见表 33-9)用于在其他 UI 元素(例如上下文菜单)之上打开窗口。所有的 Flyout 都派生自基类 FlyoutBase。FlyoutBase 类定义了一个 Placement 属性，允许定义 Flyout 的位置。它可以在屏幕中居中，也可以围绕目标元素定位。

表 33-9

| 控 件 | 说 明 |
| --- | --- |
| MenuFlyout | MenuFlyout 控件用于显示菜单项的列表 |
| Flyout | Flyout 控件可以包含一个能使用 XAML 元素自定义的项 |

## 33.4 数据绑定

对于基于 XAML 的应用程序来说，数据绑定是一个极其重要的概念。数据绑定把数据从.NET 对象传递给 UI，或从 UI 传递给.NET 对象。简单对象可以绑定到 UI 元素、对象列表和 XAML 元素上。在数据绑定中，目标可以是 XAML 元素的任意依赖属性，CLR 对象的每个属性都可以是绑定源。因为 XAML 元素也提供了.NET 属性，所以每个 XAML 元素也可以用作绑定源。图 33-20 显示了绑定源和绑定目标之间的连接。绑定定义了该连接。

Binding 对象支持源与目标之间的几种绑定模式。绑定可以是单向的，即从源信息指向目标，但如果用户在用户界面上修改了该信息，则源不会更新。要更新源，需要双向绑定。

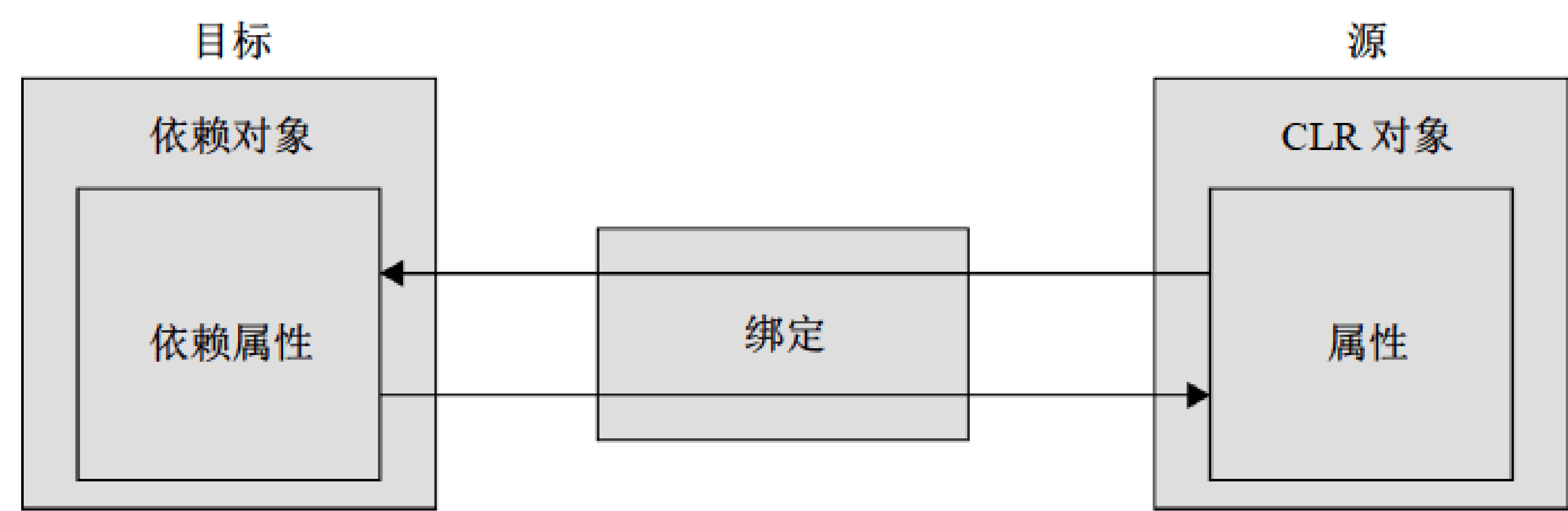



图 33-20

表 33-10 列出了绑定模式及其要求。

表 33-10

| 绑 定 模 式 | 说　明 |
|---|---|
| 一次性 | 绑定从源指向目标，且仅在应用程序启动时，或数据上下文改变时绑定一次。通过这种模式可以获得数据的快照 |
| 单向 | 绑定从源指向目标。这对于只读数据很有用，因为它不能从用户界面中修改数据。要更新用户界面，源必须实现 INotifyPropertyChanged 接口 |
| 双向 | 在双向绑定中，用户可以从 UI 中修改数据。绑定是双向的——从源指向目标，从目标指向源。源对象需要实现读/写属性，才能把改动的内容从 UI 更新到源对象上 |
| 指向源的单向 | 采用这种绑定模式，如果目标属性改变，源对象也会更新。这种绑定不能用于 UWP，但可以用于 WPF 和 Xamarin |

**注意：**

UWP 支持两种绑定类型：使用 Binding 标记扩展的传统绑定，以及使用 x:Bind 标记扩展的新编译绑定。请注意，绑定模式的默认值在这些绑定类型之间存在差异，因此最好总是指定绑定模式。本节主要关注新的编译绑定。

除了绑定模式之外，数据绑定还涉及许多方面。本节详细介绍与简单的.NET 对象和列表的绑定。通过更改通知，可以使用绑定对象中的更改更新 UI。本节也将论述如何动态地选择数据模板。

下面从 DataBindingSamples 示例应用程序开始。该应用程序显示图书列表，并允许用户选择一本书，来查看图书细节。

### 33.4.1　用 INotifyPropertyChanged 更改通知

首先创建模型。为了在属性值变化时把更改信息传递给用户界面，必须实现 INotifyPropertyChanged 接口。为了重用此实现代码，创建实现此接口的 BindableBase 类。该接口定义了 PropertyChanged 事件处理程序，该事件在 OnPropertyChanged 方法中触发。方法 Set 用于更改属性值，并触发 PropertyChanged 事件。如果要设置的值与当前值没有不同，则不触发事件，且方法仅返回 false。只有使用不同的值时，属性才设置为新值，并触发 PropertyChanged 事件。这个方法在 C#中通过 CallerMemberName 属性来使用调用者信息。propertyName 参数通过这个属性定义为可选参数，C#编译器就会通过这个参数传递属性名，所以不需要在代码中添加硬编码字符串(代码文件 DataBindingSamples/Models/BindableBase.cs)：

```
public abstract class BindableBase : INotifyPropertyChanged
{
  public event PropertyChangedEventHandler PropertyChanged;

  public virtual bool Set<T>(ref T item, T value,
    [CallerMemberName] string propertyName = null)
```

```
    {
      if (EqualityComparer<T>.Default.Equals(item, value)) return false;
      item = value;
      OnPropertyChanged(propertyName);
      return true;
    }

    protected virtual void OnPropertyChanged(string propertyName) =>
      PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
  }
```

**注意：**
调用者信息参见第 14 章。INotifyPropertyChanged 的实现参见第 34 章。

Book 类派生自基类 BindableBase，并实现了属性 BookId、Title、Publisher 和 Authors。BookId 属性是只读的；Title 和 Publisher 使用来自基类的变更通知实现；Author 是一个只读属性，返回作者列表(代码文件 DataBindingSamples/Models/Book.cs)：

```
public class Book : BindableBase
{
  public Book(int id, string title, string publisher, params string[] authors)
  {
    BookId = id;
    Title = title;
    Publisher = publisher;
    Authors = authors;
  }

  public int BookId { get; }

  private string _title;
  public string Title
  {
    get => _title;
    set => Set(ref _title, value);
  }

  private string __publisher;
  public string Publisher
  {
    get => _publisher;
    set => Set(ref _publisher, value);
  }

  public IEnumerable<string> Authors { get; }

  public override string ToString() => Title;
}
```

## 33.4.2 创建图书列表

GetSampleBooks 方法返回应使用 Book 类的构造函数显示的图书列表(代码文件 DataBindingSamples/Services/SampleBooksService.cs)：

```
public class SampleBooksService
{
  public IEnumerable<Book> GetSampleBooks() =>
    new List<Book>()
    {
      new Book(1, "Professional C# 7 and .NET Core 2", "Wrox Press",
        "Christian Nagel"),
      new Book(2, "Professional C# 6 and .NET Core 1.0", "Wrox Press",
        "Christian Nagel"),
      new Book(3, "Professional C# 5.0 and .NET 4.5.1", "Wrox Press",
        "Christian Nagel", "Jay Glynn", "Morgan Skinner"),
```

```
      new Book(4, "Enterprise Services with the .NET Framework", "AWL",
        "Christian Nagel")
    };
}
```

现在，BooksService 类提供了 RefreshBooks、GetBook、AddBook 方法以及属性 Books。属性 Books 返回一个 ObservableCollection<Book>对象。ObservableCollection 是一个泛型类，通过实现接口 INotifyCollection-Changed 来提供更改通知(代码文件 DataBindingSamples /Services/BooksService.cs)：

```
public class BooksService
{
  private ObservableCollection<Book> _books = new ObservableCollection<Book>();

  public void RefreshBooks()
  {
    _books.Clear();
    var sampleBooksService = new SampleBooksService();
    var books = sampleBooksService.GetSampleBooks();
    foreach (var book in books)
    {
      _books.Add(book);
    }
  }

  public Book GetBook(int bookId) =>
    _books.Where(b => b.BookId == bookId).SingleOrDefault();

  public void AddBook(Book book) => _books.Add(book);

  public IEnumerable<Book> Books => _books;
}
```

**注意：**
第 11 章详细介绍了泛型类 ObservableCollection。

### 33.4.3　列表绑定

现在可以显示图书列表了。可以使用任何 ItemsSource 派生控件指定 ItemsSource 属性，绑定到列表上。下面的代码片段使用 ListView 控件将 ItemsSource 绑定到 Books 属性上。使用标记扩展 x:Bind 时，指定的第一个名称是绑定的源，Mode 参数确定了绑定模式。对于 OneWay，当消息源发生变化时， UWP 利用变更通知来更新用户界面：

```
<ListView ItemsSource="{x:Bind Books, Mode=OneWay}" Grid.Row="1" />
```

在代码隐藏文件中，指定 Books 属性以引用 BooksService 的 Books 属性(代码文件 DataBindingSamples/MainPage.xaml.cs)：

```
public sealed partial class MainPage : Page
{
  private BooksService _booksService = new BooksService();
  public MainPage()
  {
    this.InitializeComponent();
  }

  public IEnumerable<Book> Books => _booksService.Books;
}
```

### 33.4.4　把事件绑定到方法

如果没有在 BooksService 中调用 RefreshBooks 方法，列表将为空。使用 XAML 文件，会创建一个 CommandBar，其中列出两个 AppBarButton 控件。通过 AppBarButton 控件，Click 事件再次绑定到 OnRefreshBooks

和 OnRefresh 方法上(代码文件 DataBindingSamples/MainPage.xaml)：

```
<CommandBar Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="2">
  <AppBarButton Icon="Refresh" Label="Refresh"
    Click="{x:Bind OnRefreshBooks}" />
  <AppBarButton Icon="Add" Label="Add Book" Click="{x:Bind OnAddBook}" />
</CommandBar>
```

如果方法没有参数或具有事件的委托类型指定的参数，则可以将事件绑定到方法。在以下代码片段中，OnRefreshBooks 和 OnAddBook 方法声明为 void，没有参数(代码文件 DataBindingSamples/MainPage.xaml.cs)：

```
public void OnRefreshBooks()
{
  _booksService.RefreshBooks();
}

public void OnAddBook() =>
  _booksService.AddBook(new Book(GetNextBookId(),
    $"Professional C# {GetNextBookId() + 3}", "Wrox Press"));

private int GetNextBookId() => Books.Select(b => b.BookId).Max() + 1;
```

**注意：**
绑定到方法上只能使用 x:Bind 标记扩展，不能使用传统的 Binding 标记扩展。

正在运行的应用程序带有两个 AppBar 按钮，如图 33-21 所示。单击 Refresh 按钮加载图书，并显示图书标题，因为 Book 类的 ToString 方法返回标题。单击 Add 按钮会创建一个新的 book 对象，该对象会出现在列表中，因为列表的类型是 ObservableCollection。ObservableCollection 通过接口 INotifyCollectionChanged 实现了更改通知。

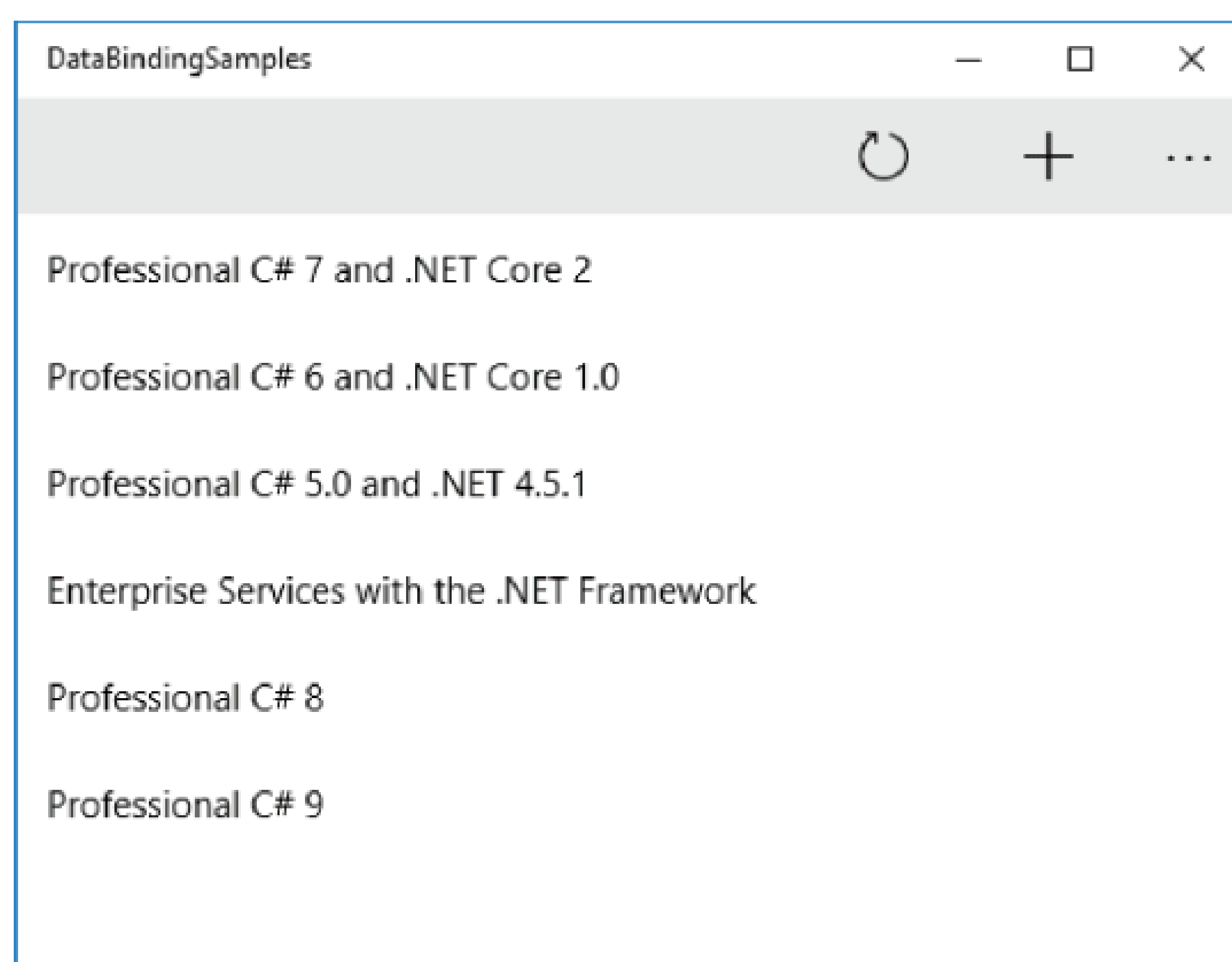



图 33-21

### 33.4.5 使用数据模板和数据模板选择器

为了创建不同的项外观，可以创建一个 DataTemplate。可以使用 x:key 特性指定的键引用 DataTemplate。使用 x:DataType 特性时，可以在数据模板中使用已编译绑定。已编译绑定需要在编译时绑定到的类型。要绑定到 Title 属性，类型由 Book 类定义(代码文件 DataBindingSamples/MainPage.xaml)：

```
<Page.Resources>
  <DataTemplate x:DataType="models:Book" x:Key="WroxTemplate">
   <Border Background="Red" Margin="4" Padding="4" BorderThickness="2"
     BorderBrush="DarkRed">
     <TextBlock Text="{x:Bind Title, Mode=OneWay}" Foreground="White"
       Width="300" />
```

```
    </Border>
  </DataTemplate>
  <!-- ... -->
</Page.Resources>
```

在 ItemsControl 中使用的数据模板可以使用 ItemsControl 的 ItemTemplate 属性来引用。现在使用 DataTemplateSelector，根据出版社的名称动态地选择 DataTemplate，而不是指定 DataTemplate。

BookDataTemplateSelector 派生自基类 DataTemplateSelector。数据模板选择器需要重写方法 SelectTemplateCore 并返回所选的 DataTemplate。在实现 BookTemplateSelector 时，指定了两个属性 WroxTemplate 和 DefaultTemplate。在 SelectTemplateCore 方法中，会接收 Book 对象。可以使用模式匹配与 switch 语句，这样，如果出版社是 Wrox Press，则返回 WroxTemplate。在其他情况下，会返回 DefaultTemplate。可以使用更多的出版社扩展 switch 语句(代码文件 DataBindingSamples/Utilities /BookTemplateSelector.cs)：

```
public class BookTemplateSelector : DataTemplateSelector
{
  public DataTemplate WroxTemplate { get; set; }
  public DataTemplate DefaultTemplate { get; set; }

  protected override DataTemplate SelectTemplateCore(object item)
  {
    DataTemplate selectedTemplate = null;

    switch (item)
    {
      case Book b when b.Publisher == "Wrox Press":
        selectedTemplate = WroxTemplate;
        break;
      default:
        selectedTemplate = DefaultTemplate;
        break;
    }
    return selectedTemplate;
  }
}
```

**注意：**
模式匹配详见第 13 章。

接下来，需要实例化和初始化数据模板选择器。可以在 XAML 代码中完成这个工作。在这里，指定属性 WroxTemplate 和 DefaultTemplate 来引用先前创建的 DataTemplate 模板(代码文件 DataBindingSamples/MainPage.xaml)：

```
<Page.Resources>
  <DataTemplate x:DataType="models:Book" x:Key="WroxTemplate">
    <Border Background="Red" Margin="4" Padding="4" BorderThickness="2"
      BorderBrush="DarkRed">
      <TextBlock Text="{x:Bind Title, Mode=OneWay}" Foreground="White"
        Width="300" />
    </Border>
  </DataTemplate>
  <DataTemplate x:DataType="models:Book" x:Key="DefaultTemplate">
    <Border Background="LightBlue" Margin="4" Padding="4" BorderThickness="2"
      BorderBrush="DarkBlue">
      <TextBlock Text="{x:Bind Title, Mode=OneWay}" Foreground="Black"
        Width="300" />
    </Border>
  </DataTemplate>
  <utils:BookTemplateSelector x:Key="BookTemplateSelector"
    WroxTemplate="{StaticResource WroxTemplate}"
    DefaultTemplate="{StaticResource DefaultTemplate}" />
</Page.Resources>
```

为了将 BookTemplateSelector 与 ListView 中的项一起使用，ItemTemplateSelector 属性使用键和 StaticResource 标记扩展来引用模板：

```
<ListView ItemsSource="{x:Bind Books, Mode=OneWay}"
  ItemTemplateSelector="{StaticResource BookTemplateSelector}"
  Grid.Row="1" />
```

在此阶段运行应用程序时，图 33-22 显示了基于出版社的不同视图的新输出。

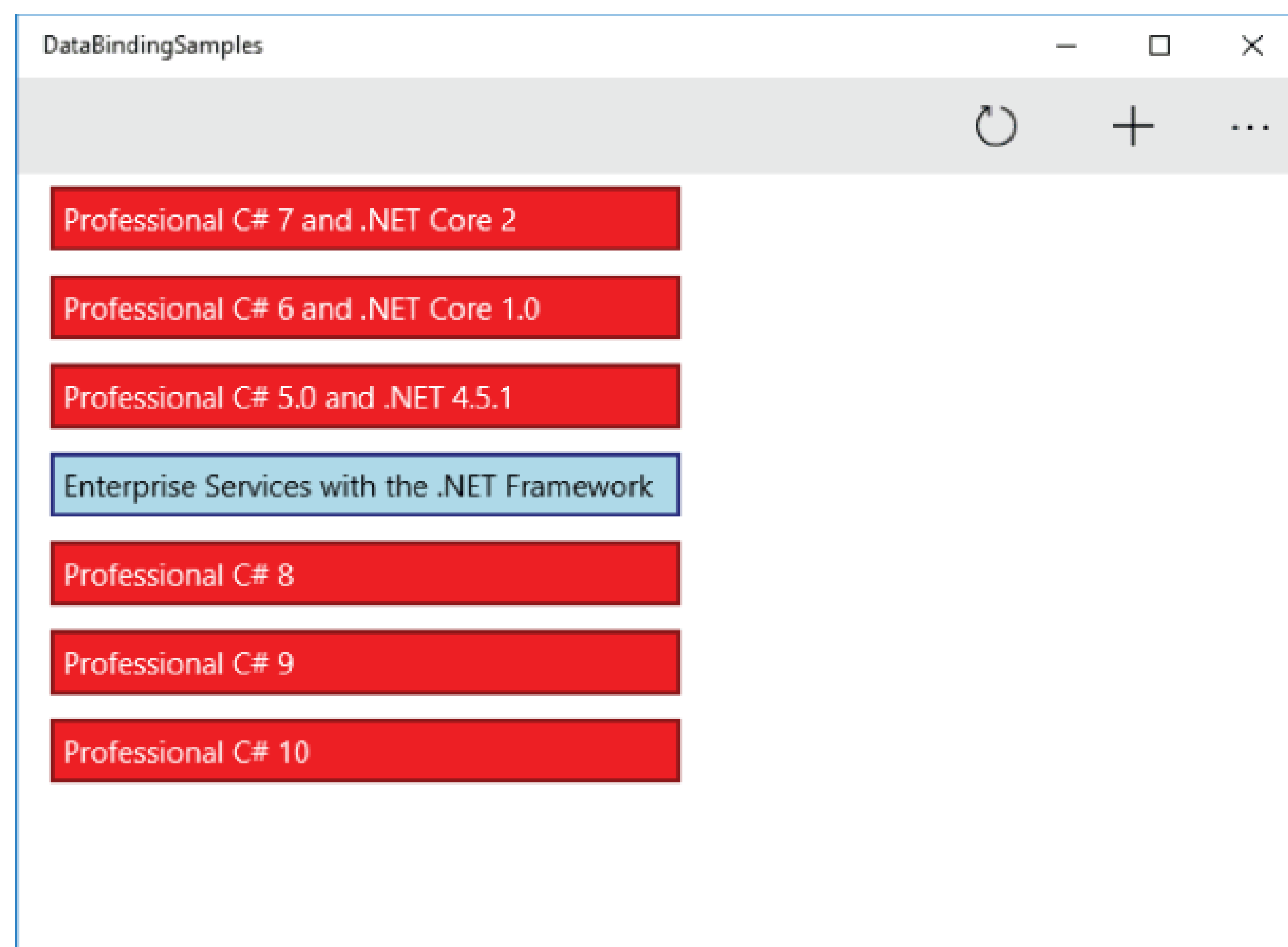



图 33-22

### 33.4.6 绑定简单对象

不只是绑定列表，单本书应该显示在应用程序的右侧。已编译绑定用于绑定 Book 对象的 BookId、Title 和 Publisher 属性(代码文件 DataBindingSamples/Views/BookUserControl.xaml)：

```
<UserControl
    x:Class="DataBindingSamples.Views.BookUserControl"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:DataBindingSamples.Views"
    xmlns:conv="using:DataBindingSamples.Converters"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    d:DesignHeight="300"
    d:DesignWidth="400">
    <!-- ... -->
    <StackPanel Orientation="Vertical" Grid.Row="1">
      <TextBox Header="BookId" IsReadOnly="True"
        Text="{x:Bind Book.BookId, Mode=OneWay}" />
      <TextBox Header="Title" Text="{x:Bind Book.Title, Mode=TwoWay}" />
      <TextBox Header="Publisher"
        Text="{x:Bind Book.Publisher, Mode=TwoWay}" />
      <!-- ... -->
    </StackPanel>
  </Grid>
</UserControl>
```

在代码隐藏文件中，Book 属性定义为一个依赖属性。当值更改时，需要更改通知来进行更新，这就是为什么要使用依赖属性的原因。还可以实现 INotifyPropertyChanged，但是由于依赖属性已经可以从基类 DependencyObject 中获得，所以可以轻松地使用依赖属性(代码文件 DataBindingSamples/Views/BookUser Control.xaml.cs)：

```
public Book Book
{
  get => (Book)GetValue(BookProperty);
  set => SetValue(BookProperty, value);
}
```

```
public static readonly DependencyProperty BookProperty =
  DependencyProperty.Register("Book", typeof(Book), typeof(BookUserControl),
    new PropertyMetadata(null));
```

现在，用户控件需要显示在 MainPage 中，当前选中的图书应该分配给用户控件的 Book 属性。为此可以使用 XAML 代码。BookUserControl 在 MainPage 的 Grid 中添加，在 ListView 中，SelectedItem 属性绑定到 Book 属性。这次， TwoWay 绑定需要在 ListView 中更新 UserControl(代码文件 DataBindingSamples/MainPage.xaml)：

```
<ListView x:Name="BooksList" ItemsSource="{x:Bind Books, Mode=OneWay}"
  ItemTemplateSelector="{StaticResource BookTemplateSelector}"
  SelectedItem="{x:Bind CurrentBook.Book, Mode=TwoWay}" Grid.Row="1" />
  <views:BookUserControl x:Name="CurrentBook" Grid.Row="1" Grid.Column="1"
    Margin="4" />
```

**注意：**

*也可以以另一种方式创建绑定——将 BookUserControl 绑定到 ListView。这样，OneWay 绑定就足够了——只需要将更新后的值从 ListView 获取到 BookUserControl。但是在这里 XAML 编译器会报错，因为它不能将一个对象(来自 ListView)分配给 BookUserControl 的强类型 Book 属性。可以通过创建一个值转换器(稍后讨论)来解决这个问题。SelectedItem 属性不存在这个问题，因为微软改变了实现，最新的 Windows 10 版本不再报错。在早期版本中，还需要在该场景中使用对象到对象的转换器。*

在运行应用程序时，可以看到图书，在列表中选择一本书时，详细信息显示在用户控件中，如图 33-23 所示。

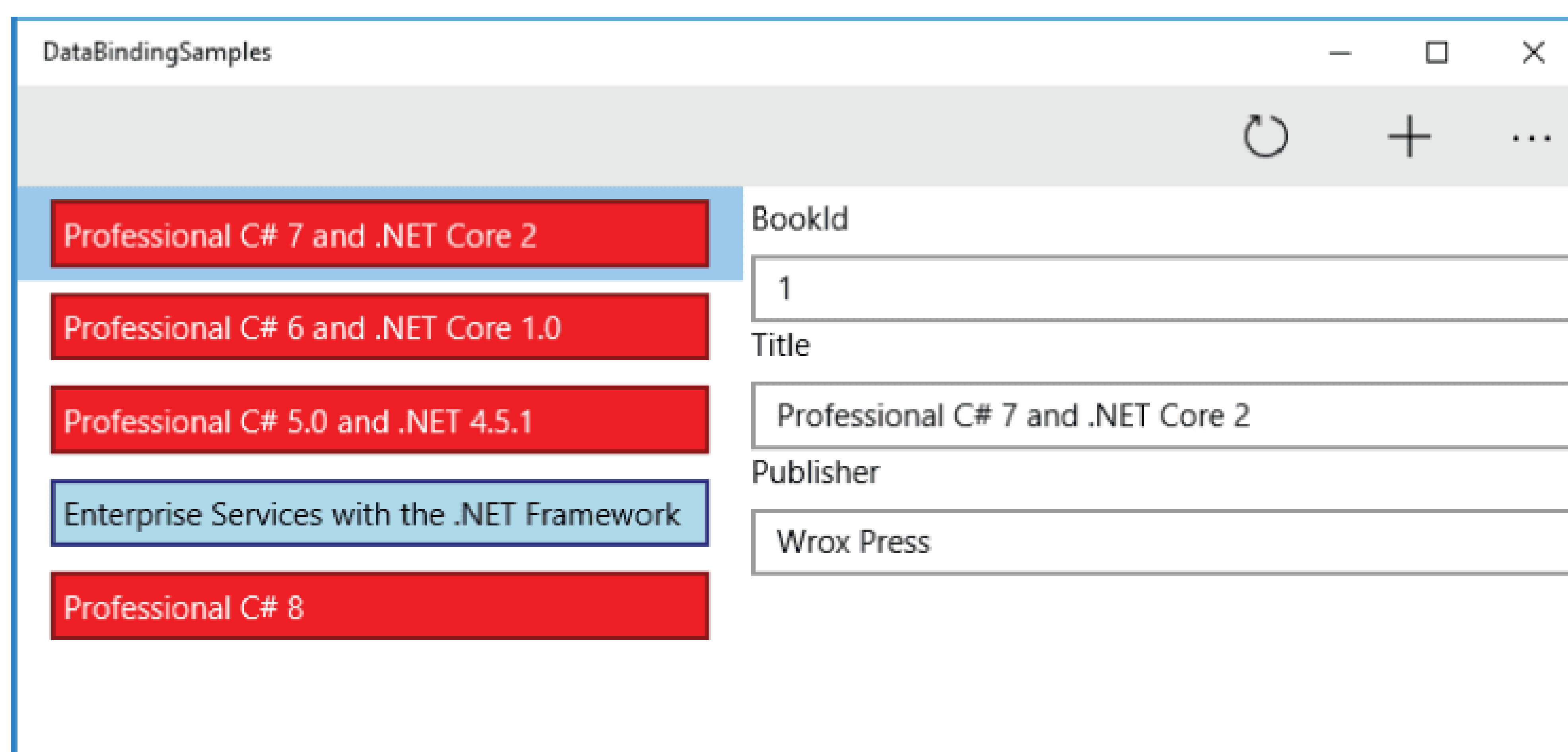



图 33-23

### 33.4.7 值的转换

作者还没有显示在用户控件中。原因是 Authors 属性是一个列表。可以在用户控件中定义一个 ItemsControl 来显示 Authors 属性。但是，仅为了显示一个以逗号分隔的作者列表，使用 TextBlock 即可。只需要一个转换器就可以将 IEnumerable<string>(Authors 属性的类型)转换为字符串。

值转换器是 IValueConverter 接口的实现。这个接口定义了 Convert 和 ConvertBack 方法。对于双向绑定，需要实现这两个方法。使用单向绑定，Convert 方法就足够了。类 CollectionToStringConverter 使用 string.Join 方法创建单个字符串，实现了 Convert 方法。值转换器还接收一个对象 parameter，可以在使用值转换器时指定该参数。这里，将该参数用作字符串分隔符(代码文件 DataBindingSamples/Converters/CollectionToStringConverter.cs)：

```
public class CollectionToStringConverter : IValueConverter
{
  public object Convert(object value, Type targetType, object parameter,
    string language)
  {
    IEnumerable<string> names = (IEnumerable<string>)value;
    return string.Join(parameter?.ToString() ?? ", ", names);
  }
```

```
    public object ConvertBack(object value, Type targetType, object parameter,
      string language)
    {
      throw new NotImplementedException();
    }
  }
```

使用用户控件，CollectionToStringConverter 在资源部分实例化(代码文件 DataBindingSamples/Views/BookUserControl.xaml)：

```
<UserControl.Resources>
  <conv:CollectionToStringConverter x:Key="CollectionToStringConverter" />
</UserControl.Resources>
```

现在可以使用 Converter 属性在 x:Bind 标记扩展中引用转换器。ConverterParameter 属性指定在之前的 string.Join 方法中使用的字符串分隔符 (代码文件 DataBindingSamples/Views/BookUserControl.xaml)：

```
<TextBox Header="Authors" IsReadOnly="True"
  Text="{x:Bind Book.Authors, Mode=OneWay,
    Converter={StaticResource CollectionToStringConverter},
    ConverterParameter='; '}" />
```

运行该应用程序时，作者将如图 33-24 所示。

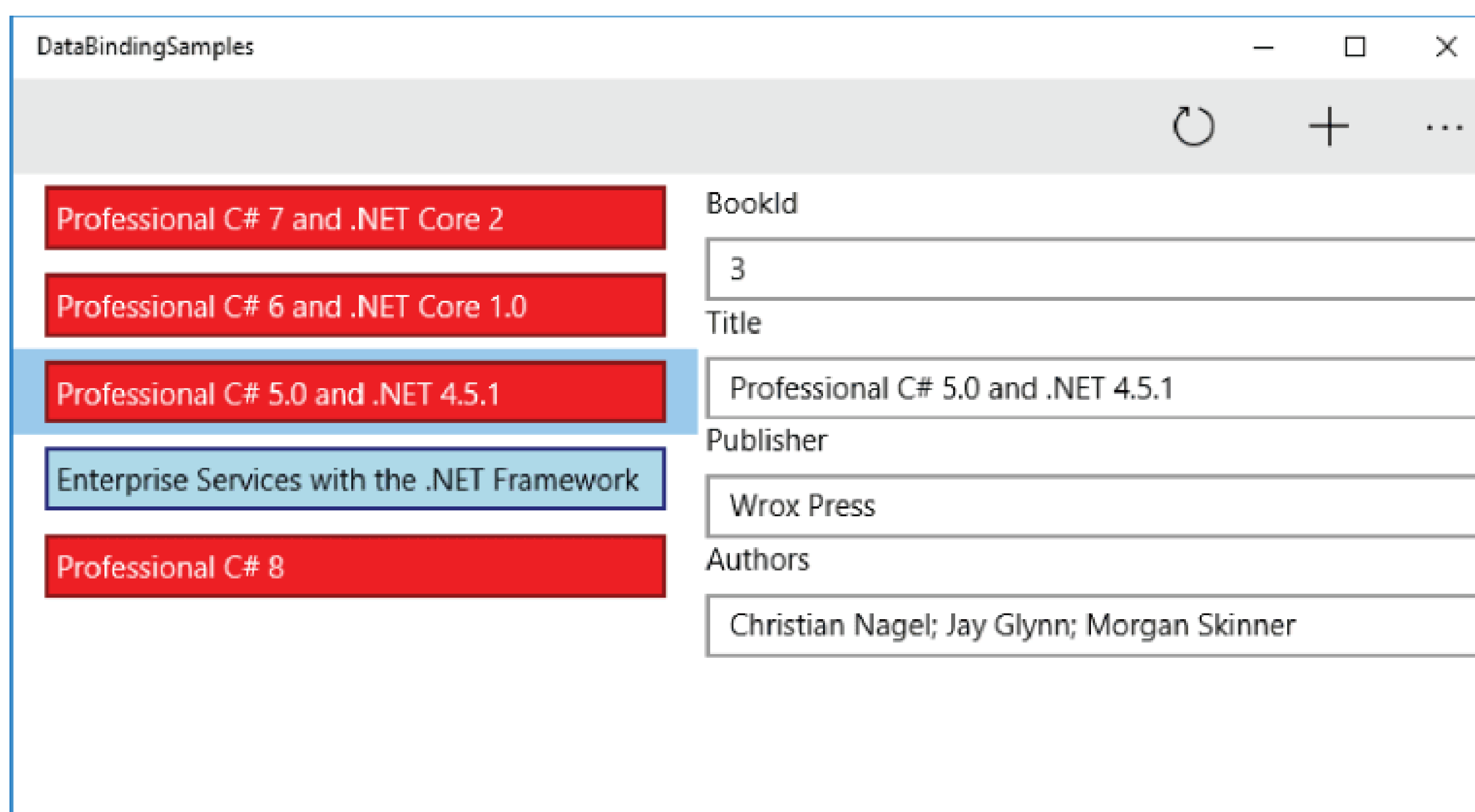



图 33-24

**注意：**

第 36 章介绍了使用已编译数据绑定的更多特性，例如使用绑定生命周期、在资源文件中的编译绑定和阶段绑定。

# 33.5 导航

如果应用程序是由多个页面组成的，就需要能在这些页面之间导航。有不同的应用程序结构需要导航，比如使用汉堡包按钮导航到不同的根页面，或者使用不同的选项卡和替换选项卡项。

如果需要为用户提供导航的方法，导航的核心是 Frame 类。Frame 类允许使用 Navigate 方法，选择性地传递参数，导航到具体的页面上。Frame 类有一个要导航的页面堆栈，因此可以后退、前进，限制堆栈中页面的数量等。

导航的一个重要方面是能够返回。下面几节介绍了使用回航的 Windows 10 方法。

## 33.5.1 导航回最初的页面

下面开始创建一个有多个页面的 Windows 应用程序，在页面之间导航。模板生成的代码在 App 类中包含

OnLaunched 方法，在该方法中，实例化一个 Frame 对象，再调用 Navigate 方法，导航到 MainPage (代码文件 PageNavigation/App.xaml.cs)：

```
protected override void OnLaunched(LaunchActivatedEventArgs e)
{
  Frame rootFrame = Window.Current.Content as Frame;
  if (rootFrame == null)
  {
    rootFrame = new Frame();
    rootFrame.NavigationFailed += OnNavigationFailed;
    if (e.PreviousExecutionState == ApplicationExecutionState.Terminated)
    {
      //TODO: Load state from previously suspended application
    }
    Window.Current.Content = rootFrame;
  }
  if (rootFrame.Content == null)
  {
    rootFrame.Navigate(typeof(MainPage), e.Arguments);
  }
  Window.Current.Activate();
}
```

**注意：**
源代码有一个 TODO 注释，从前面暂停的应用程序中加载状态。如何处理暂停在第 36 章中解释。

Frame 类有一个已访问的页面堆栈。GoBack 方法可以在这个堆栈中回航(如果 CanGoBack 属性返回 true)，GoForward 方法可以在后退后前进到下一页。Frame 类还提供了几个导航事件，如 Navigating、Navigated、NavigationFailed 和 NavigationStopped。

为了查看导航操作，除了 MainPage 之外，还创建 SecondPage 和 ThirdPage 页面，在这些页面之间导航。在 MainPage 上，可以导航到 SecondPage，通过传递一些数据可以从 SecondPage 导航到 ThirdPage。

因为有这些页面之间的通用功能，所以创建一个基类 NavigationPage，所有这些页面都派生自它。NavigationPage 类派生自基类 Page，实现了接口 INotifyPropertyChanged，用于更新用户界面(代码文件 PageNavigation/NavigationPage.cs)：

```
public abstract class NavigationPage : Page, INotifyPropertyChanged
{
  public event PropertyChangedEventHandler PropertyChanged;

  protected virtual void OnPropertyChanged(
    [CallerMemberName] string propertyName = null) =>
      PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));

  protected bool SetProperty<T>(ref T item, T value,
    [CallerMemberName] string propertyName = null)
  {
    if (EqualityComparer<T>.Default.Equals(item, value)) return false;
    item = value;
    OnPropertyChanged(propertyName);
    return true;
  }

  private string _navigationMode;
  public string NavigationMode
  {
    get => _navigationMode;
    set => SetProperty(ref _navigationMode, value);
  }

  //...
}
```

### 33.5.2　重写 Page 类的导航

Page 类是 NavigationPage 的基类(也是 XAML 页面的基类)，该类定义了用于导航的方法。当导航到相应的

页面时，会调用 OnNavigatedTo 方法。在这个页面中，可以看到导航是如何操作的(NavigationMode 属性)和导航参数。OnNavigatingFrom 方法是从页面中退出时调用的第一个方法。在这里，导航可以取消。从这个页面中退出时，最终调用的是 OnNavigatedFrom 方法。在这里，应该清理 OnNavigatedTo 方法分配的资源(代码文件 PageNavigation/App.xaml.cs)：

```
public abstract class NavigationPage : Page, INotifyPropertyChanged
{
  //...
  protected override void OnNavigatedTo(NavigationEventArgs e)
  {
    base.OnNavigatedTo(e);
    NavigationMode = $"Navigation Mode: {e.NavigationMode}";
    //...
  }

  protected override void OnNavigatingFrom(NavigatingCancelEventArgs e)
  {
    base.OnNavigatingFrom(e);
  }

  protected override void OnNavigatedFrom(NavigationEventArgs e)
  {
    base.OnNavigatedFrom(e);
  }
}
```

### 33.5.3 在页面之间导航

下面实现 3 个页面。为了使用 NavigationPage 类，代码隐藏文件需要修改，以使用 NavigationPage 作为基类(代码文件 PageNavigation/MainPage.xaml.cs)：

```
public sealed partial class MainPage : NavigationPage
{
  //...
}
```

基类的变化也需要反映在 XAML 文件中：使用 NavigationPage 元素代替 Page (代码文件 PageNavigation/MainPage.xaml)：

```
<local:NavigationPage
  x:Class="PageNavigation.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:PageNavigation"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  mc:Ignorable="d">
```

MainPage 包含一个 TextBlock 元素和一个 Button 控件，TextBlock 元素绑定到 BasePage 中声明的 NavigationMode 属性上，按钮的 Click 事件绑定到 OnNavigateToSecondPage 方法上(代码文件 PageNavigation/MainPage.xaml)：

```
<StackPanel Orientation="Vertical">
  <TextBlock Style="{StaticResource TitleTextBlockStyle}" Margin="8">
    Main Page</TextBlock>
  <TextBlock Text="{x:Bind NavigationMode, Mode=OneWay}" Margin="8" />
  <Button Content="Navigate to SecondPage" Click="OnNavigateToSecondPage"
    Margin="8" />
</StackPanel>
```

处理程序方法 OnNavigateToSecondPage 使用 Frame.Navigate 导航到 SecondPage。Frame 是 Page 类上返回 Frame 实例的一个属性(代码文件 PageNavigation/MainPage.xaml.cs)：

```
public void OnNavigateToSecondPage()
{
  Frame.Navigate(typeof(SecondPage));
}
```

当从 SecondPage 导航到 ThirdPage 时，把一个参数传递给目标页面。参数可以在绑定到 Data 属性的文本框中输入(代码文件 PageNavigation /SecondPage.xaml)：

```
<StackPanel Orientation="Vertical">
  <TextBlock Style="{StaticResource TitleTextBlockStyle}" Margin="8">
    Second Page</TextBlock>
  <TextBlock Text="{x:Bind NavigationMode, Mode=OneWay}" Margin="8" />
  <TextBox Header="Data" Text="{x:Bind Data, Mode=TwoWay}" Margin="8" />
  <Button Content="Navigate to Third Page"
    Click="{x:Bind OnNavigateToThirdPage, Mode=OneTime}" Margin="8" />
</StackPanel>
```

在代码隐藏文件中，将 Data 属性传递给 Navigate 方法(代码文件 PageNavigation /SecondPage.xaml.cs)：

```
public string Data { get; set; }

public void OnNavigateToThirdPage()
{
  Frame.Navigate(typeof(ThirdPage), Data);
}
```

接收到的参数在 ThirdPage 中检索。在 OnNavigatedTo 方法中，NavigationEventArgs 用 Parameter 属性接收参数。Parameter 属性是 object 类型，可以给页面导航传递任何数据(代码文件 PageNavigation/ThirdPage.xaml.cs)：

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
  base.OnNavigatedTo(e);
  Data = e.Parameter as string;
}

private string _data;
public string Data
{
  get => _data;
  set => SetProperty(ref _data, value);
}
```

### 33.5.4　后退按钮

当应用程序中有导航要求时，必须包括返回的方式。在 Windows 8 中，定制的后退按钮位于页面的左上角。在 Windows 10 中仍然可以这样做。的确，一些微软应用程序包括这样一个按钮，Microsoft Edge 在左上角放置了后退和前进按钮。应在前进按钮的附近放置后退按钮。在 Windows10 中，可以利用系统的后退按钮。

根据应用程序运行在桌面模式还是平板电脑模式，后退按钮位于不同的地方。要启用这个后退按钮，需要把 SystemNavigationManager 的 AppViewBackButtonVisibility 设置为 AppViewBackButton Visibility，在下面的代码中，Frame.CanGoBack 属性返回 true 时，就是这种情况(代码文件 PageNavigation/NavigationPage.cs)：

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
  NavigationMode = $"Navigation Mode: {e.NavigationMode}";
  SystemNavigationManager.GetForCurrentView().AppViewBackButtonVisibility =
    Frame.CanGoBack ? AppViewBackButtonVisibility.Visible :
    AppViewBackButtonVisibility.Collapsed;
  base.OnNavigatedTo(e);
}
```

接下来，使用 SystemNavigationManager 类的 BackRequested 事件。对 BackRequestedEvent 的响应可以用于完整的应用程序，如这里所示。如果只在几页上需要这个功能，还可以把这段代码放在页面的 OnNavigatedTo 方法中(代码文件 PageNavigation/App.xaml.cs)：

```
protected override void OnLaunched(LaunchActivatedEventArgs e)
{
  //...
  SystemNavigationManager.GetForCurrentView().BackRequested +=
    App_BackRequested;
  Window.Current.Activate();
}
```

处理程序方法 App_BackRequested 在 frame 对象上调用 GoBack 方法(代码文件 PageNavigation/App.xaml.cs)：

```
private void App_BackRequested(object sender, BackRequestedEventArgs e)
{
  Frame rootFrame = Window.Current.Content as Frame;
  if (rootFrame == null) return;
  if (rootFrame.CanGoBack && e.Handled == false)
  {
    e.Handled = true;
    rootFrame.GoBack();
  }
}
```

在桌面模式中运行这个应用程序时，可以看到后退按钮位于上边界的左边角落里(见图 33-25)。如果应用程序在平板模式下运行，边界是不可见的，但后退按钮显示在底部边界 Windows 按钮的旁边(见图 33-26)。这是应用程序的新后退按钮。如果应用程序不能导航，用户按下后退按钮，就导航回以前的应用程序。

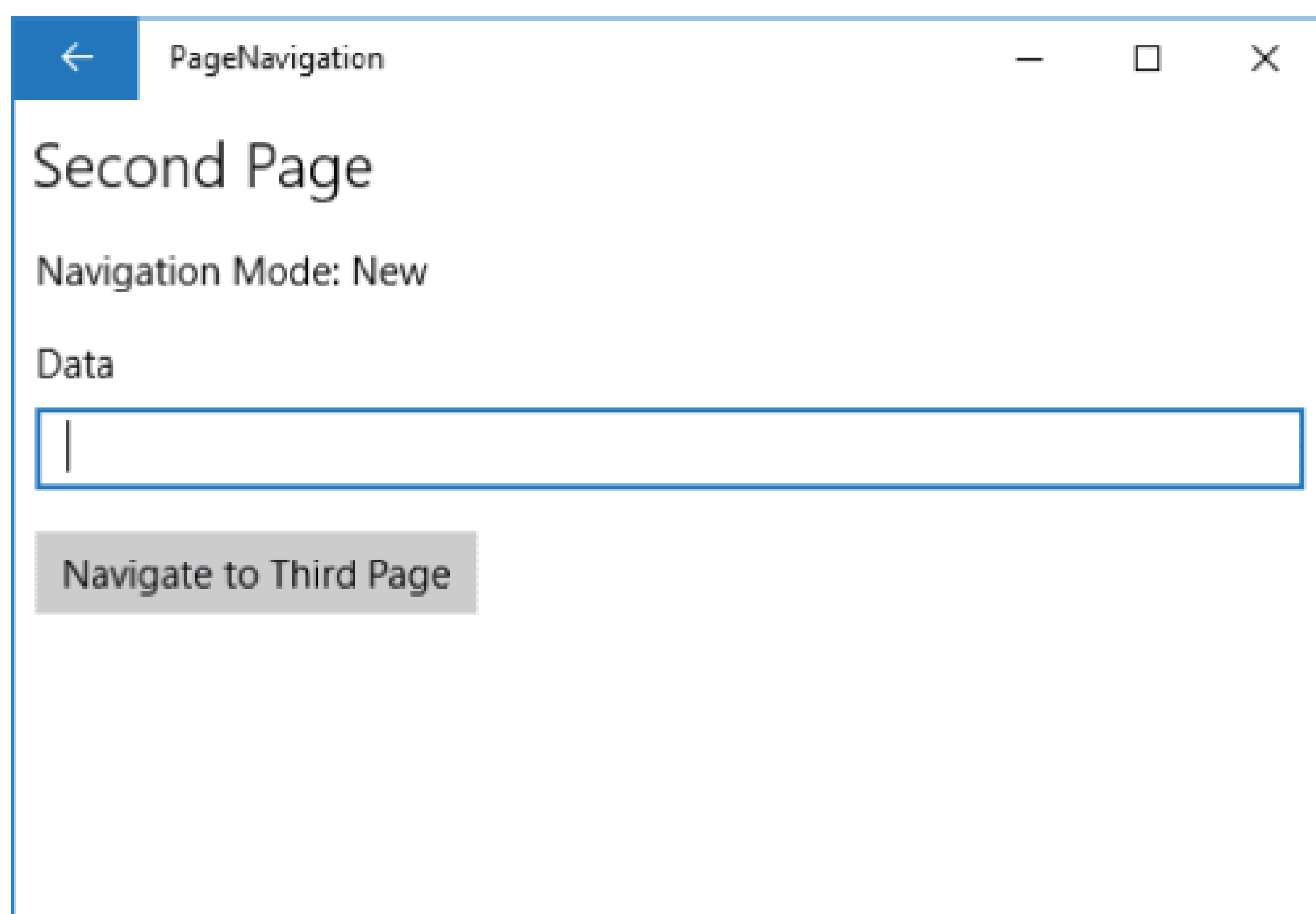


图 33-25


图 33-26

## 33.5.5 Hub

也可以让用户使用 Hub 控件在单个页面的内容之间导航。这里可以使用的一个例子是，希望显示一个图像，作为应用程序的入口点，用户滚动时显示更多的信息(参见图 33-27 中 Microsoft Store 的照片搜索应用程序)。

使用 Hub 控件可以定义多个部分。每个部分有标题和内容。也可以让标题可以单击，例如，导航到详细信息页面上。以下代码示例定义了一个 Hub 控件，在其中可以单击部分 2 和部分 3 的标题。单击某部分的标题时，就调用 Hub 控件的 SectionHeaderClick 事件指定的方法。每个部分都包括一个标题和一些内容。部分的内容由 DataTemplate 定义(代码文件 NavigationControls / HubPage.xaml)：

```
<Hub Background="{ThemeResource ApplicationPageBackgroundThemeBrush}"
  SectionHeaderClick="{x:Bind OnHeaderClick}">
  <Hub.Header>
    <StackPanel Orientation="Horizontal">
      <TextBlock>Hub Header</TextBlock>
      <TextBlock Text="{x:Bind Info, Mode=TwoWay}" />
    </StackPanel>
  </Hub.Header>
  <HubSection Width="400" Background="LightBlue" Tag="Section 1">
    <HubSection.Header>
      <TextBlock>Section 1 Header</TextBlock>
    </HubSection.Header>
    <DataTemplate>
      <TextBlock>Section 1</TextBlock>
    </DataTemplate>
  </HubSection>
  <HubSection Width="300" Background="LightGreen" IsHeaderInteractive="True"
    Tag="Section 2">
    <HubSection.Header>
      <TextBlock>Section 2 Header</TextBlock>
    </HubSection.Header>
```

```
    <DataTemplate>
      <TextBlock>Section 2</TextBlock>
    </DataTemplate>
  </HubSection>
  <HubSection Width="300" Background="LightGoldenrodYellow"
    IsHeaderInteractive="True" Tag="Section 3">
    <HubSection.Header>
      <TextBlock>Section 3 Header</TextBlock>
    </HubSection.Header>
    <DataTemplate>
      <TextBlock>Section 3</TextBlock>
    </DataTemplate>
  </HubSection>
</Hub>
```

单击标题部分时，Info 依赖属性就指定 Tag 属性的值。Info 属性绑定在 Hub 控件的标题上(代码文件 NavigationControls /HubPage.xaml.cs)：

```
public void OnHeaderClick(object sender, HubSectionHeaderClickEventArgs e)
{
  Info = e.Section.Tag as string;
}

public string Info
{
  get => (string)GetValue(InfoProperty);
  set => SetValue(InfoProperty, value);
}

public static readonly DependencyProperty InfoProperty =
  DependencyProperty.Register("Info", typeof(string), typeof(HubPage),
    new PropertyMetadata(string.Empty));
```

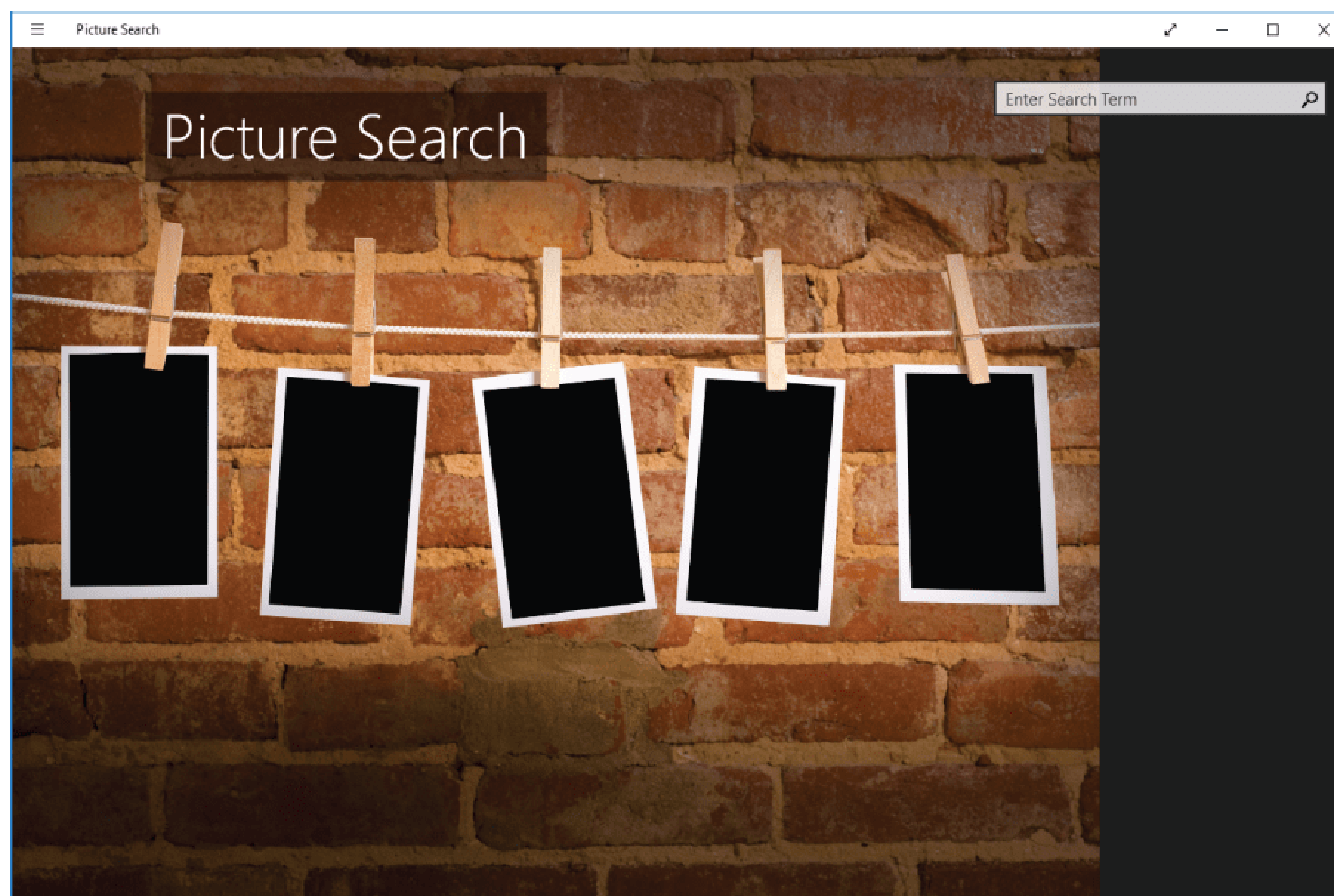



图 33-27

运行这个应用程序时，可以看到多个 hub 部分(参见图 33-28)，在部分 2 和部分 3 上有 See More 链接，因为在这些部分中，将 IsHeaderInteractive 设置为 true。当然，可以创建一个定制的标题模板，给标题指定不同的外观。

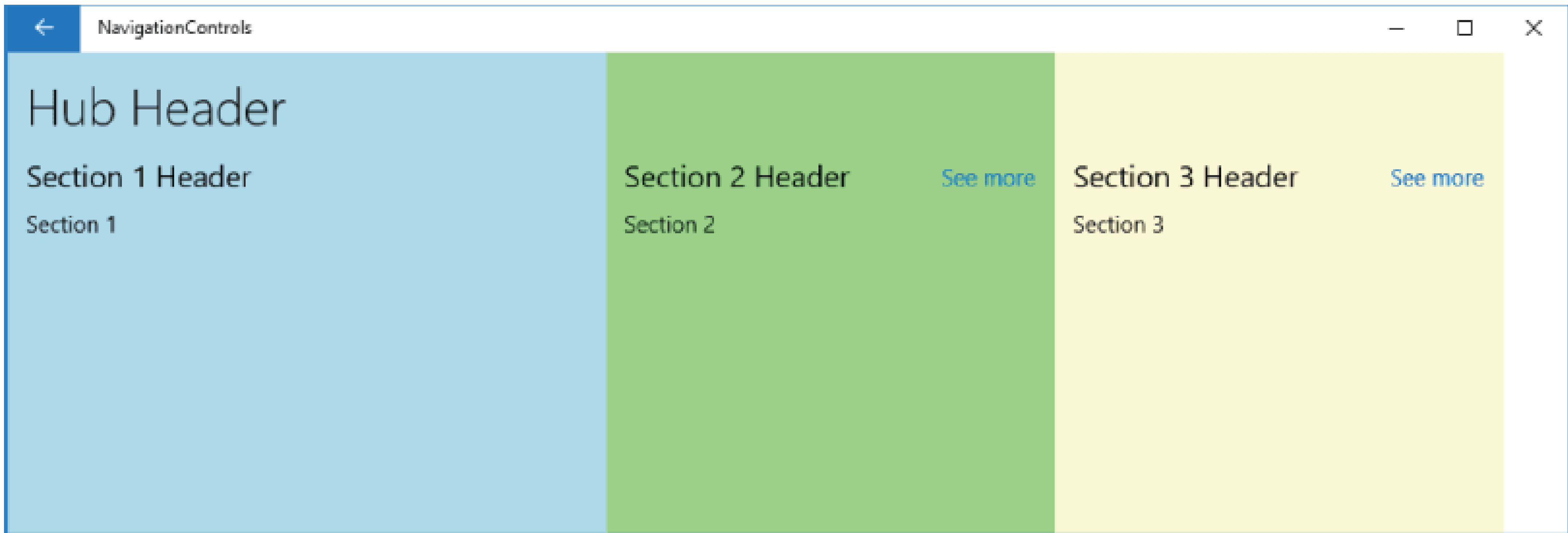



图 33-28

**注意：**
创建自定义模板参见第 35 章。

### 33.5.6 Pivot

使用 Pivot 控件可以为导航创建类似枢轴的外观。Pivot 控件可以包含多个 PivotItem 控件。每个 PivotItem 控件都有一个标题和内容。Pivot 本身包含左、右标题。示例代码填充了右标题(代码文件 NavigationControls/PivotPage.xaml)：

```
<Pivot Title="Pivot Sample"
  Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
  <Pivot.RightHeader>
    <StackPanel>
      <TextBlock>Right Header</TextBlock>
    </StackPanel>
  </Pivot.RightHeader>
  <PivotItem>
    <PivotItem.Header>Header Pivot 1</PivotItem.Header>
    <TextBlock>Pivot 1 Content</TextBlock>
  </PivotItem>
  <PivotItem>
    <PivotItem.Header>Header Pivot 2</PivotItem.Header>
    <TextBlock>Pivot 2 Content</TextBlock>
  </PivotItem>
  <PivotItem>
    <PivotItem.Header>Header Pivot 3</PivotItem.Header>
    <TextBlock>Pivot 3 Content</TextBlock>
  </PivotItem>
  <PivotItem>
    <PivotItem.Header>Header Pivot 4</PivotItem.Header>
    <TextBlock>Pivot 4 Content</TextBlock>
  </PivotItem>
</Pivot>
```

运行应用程序时，可以看到 Pivot 控件(参见图 33-29)。右标题在右边总是可见。单击一个标题，可以查看项的内容。

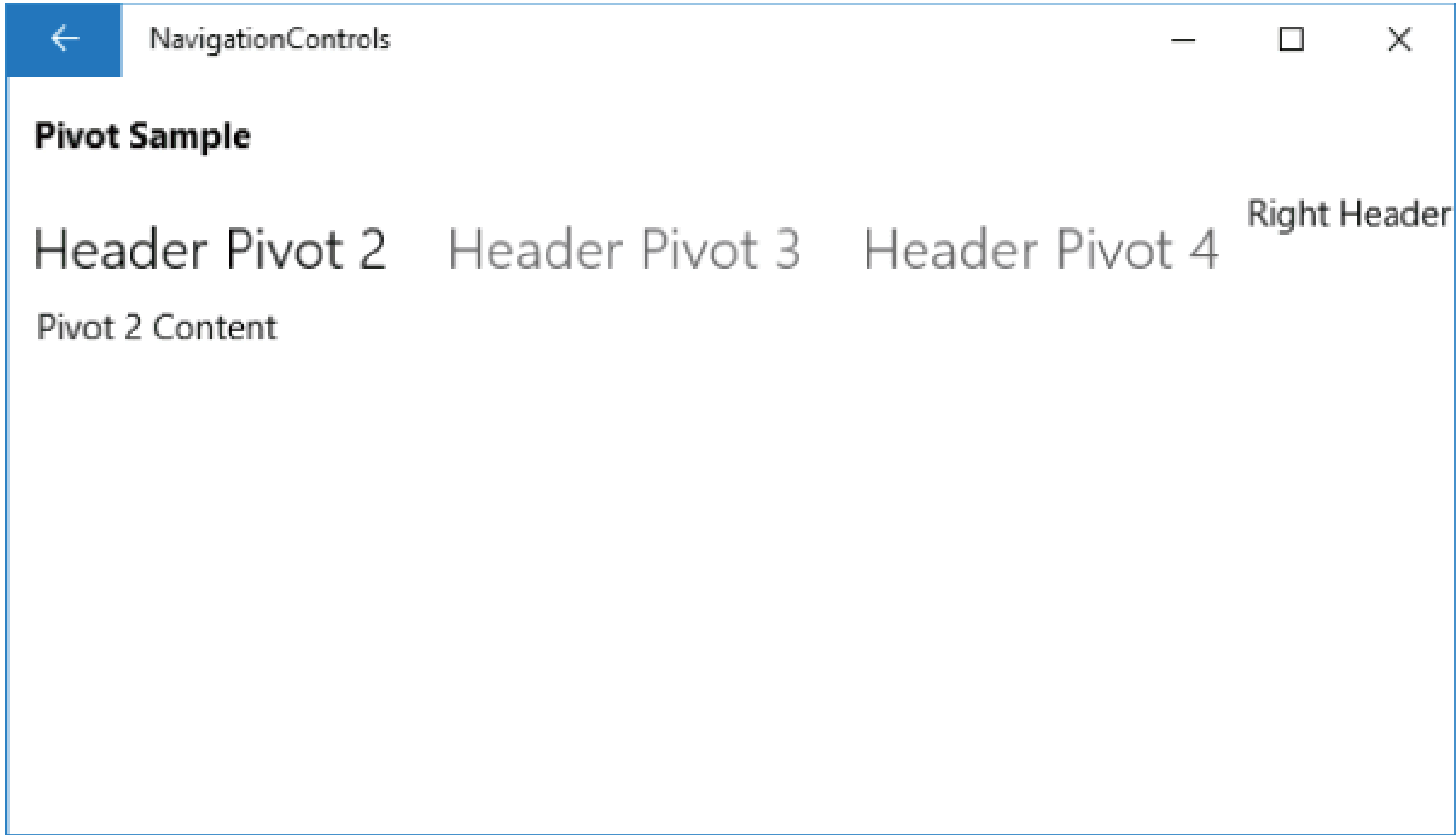



图 33-29

如果所有标题不符合屏幕的大小，用户就可以滚动。使用鼠标进行导航，可以看到左右边的箭头，如图 33-30 所示。

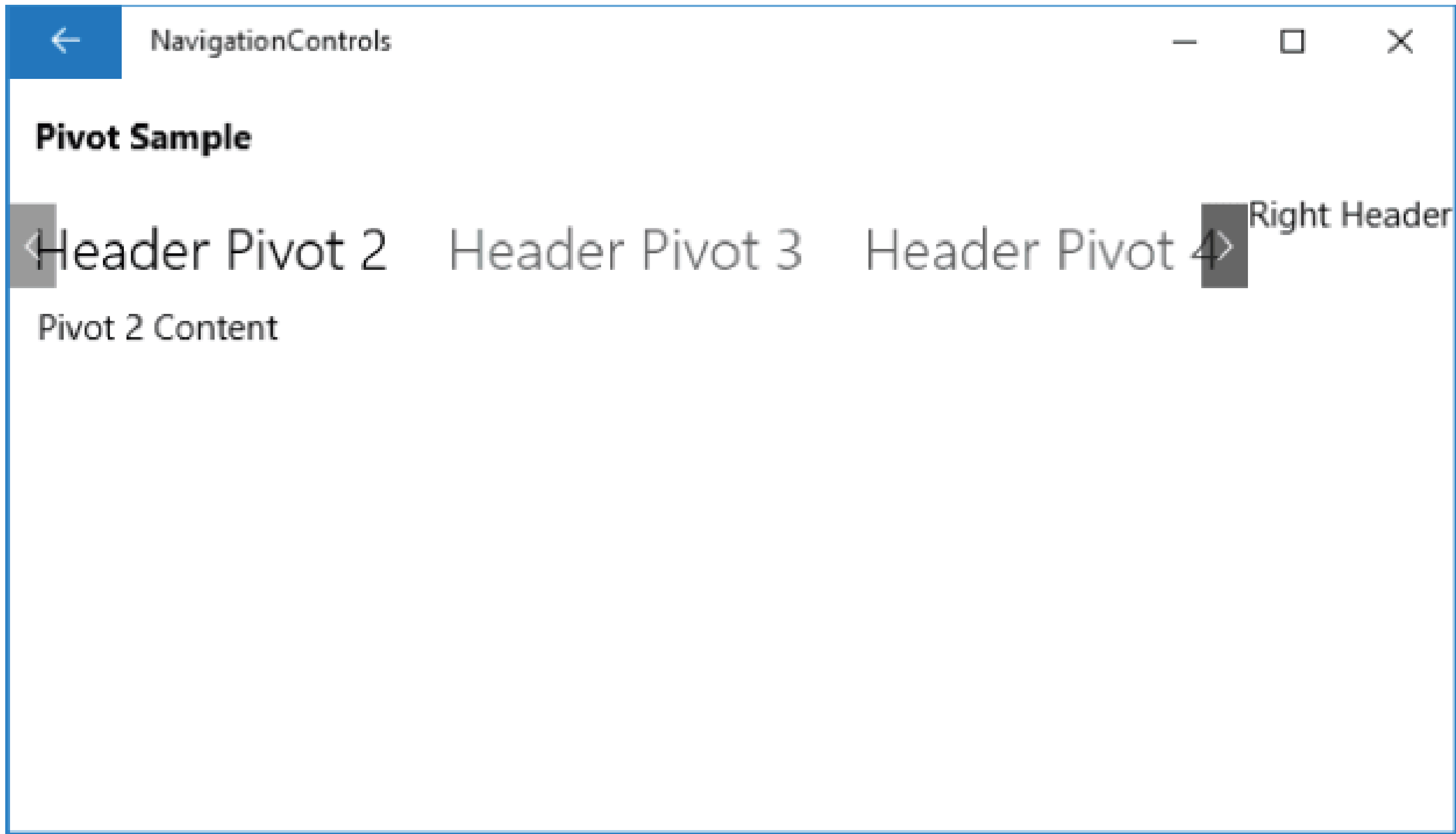



图 33-30

## 33.5.7　NavigationView

Windows 10 应用程序通常使用 SplitView 控件和汉堡包按钮。汉堡包按钮用于打开菜单列表。菜单会显示为一个图标，如果有更多的可用空间，菜单就显示图标和文本。为了给内容和菜单安排空间，SplitView 控件开始发挥作用。SplitView 为窗格和内容提供了空间，其中窗格通常包含菜单项。窗格可以有一个小尺寸和一个大尺寸，可以根据可用的屏幕大小对其进行配置。

在 Windows 10 构建号 16299 之前，必须使用 SplitView、汉堡包按钮和显示在窗格中的菜单列表，手工构建用户界面。从本书前一版《C#高级编程(第 10 版)　C#6 & .NET Core 1.0》的代码下载中可以获得一个示例，如果需要支持构建号 16299 之前的 Windows 10 版本，这是必需的。在构建号 16299 版本中，可以使用 NavigationView 控件。NavigationView 将所有这些行为集成到一个控件中。图 33-31 显示了打开的 NavigationView 窗格。单击汉堡包按钮或缩小应用程序，将窗格更改为紧凑模式，如图 33-32 所示。进一步减小应用程序的大小，将 NavigationView 的左侧部分减少为汉堡包按钮，如图 33-33 所示。

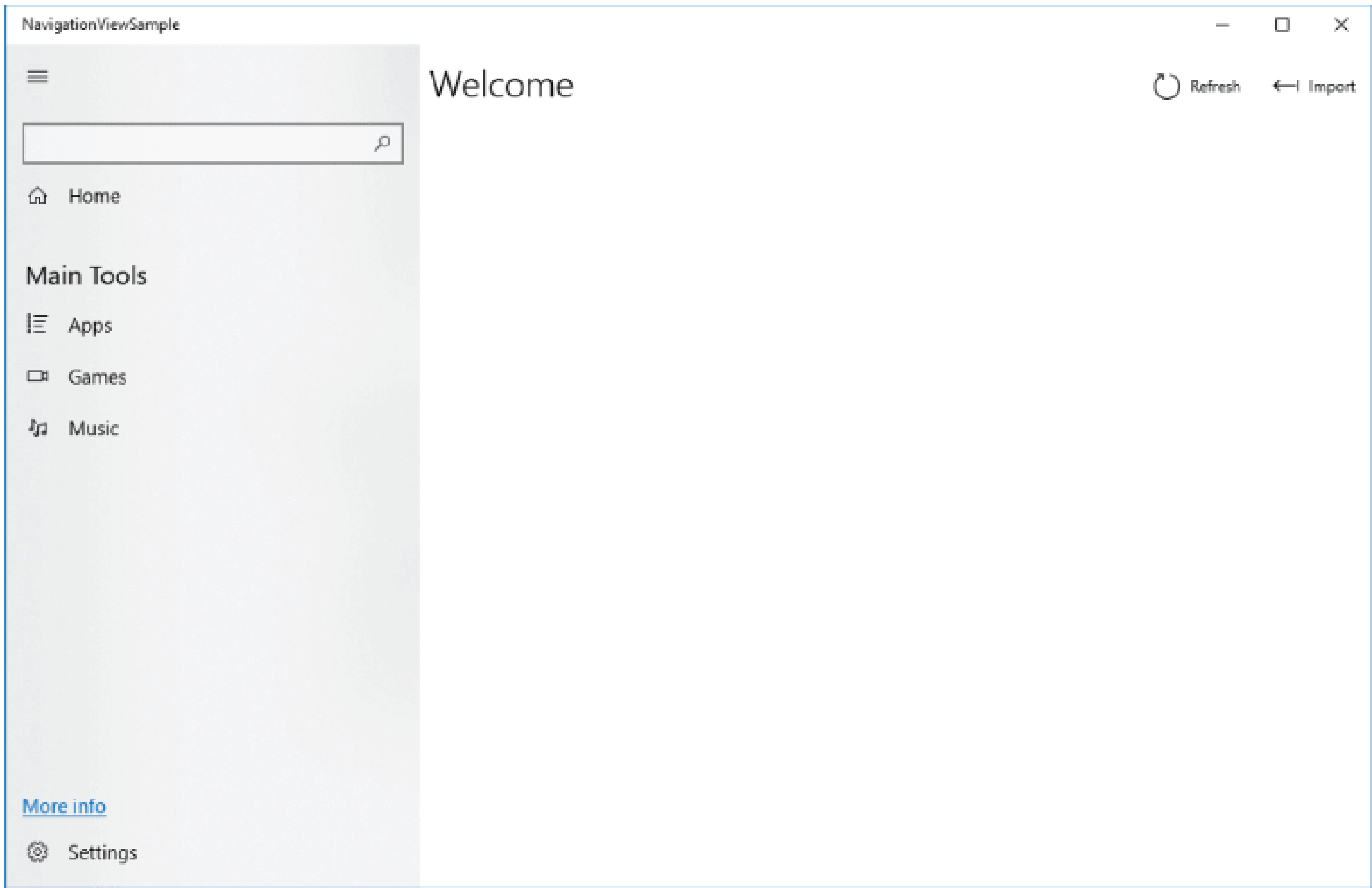



图 33-31

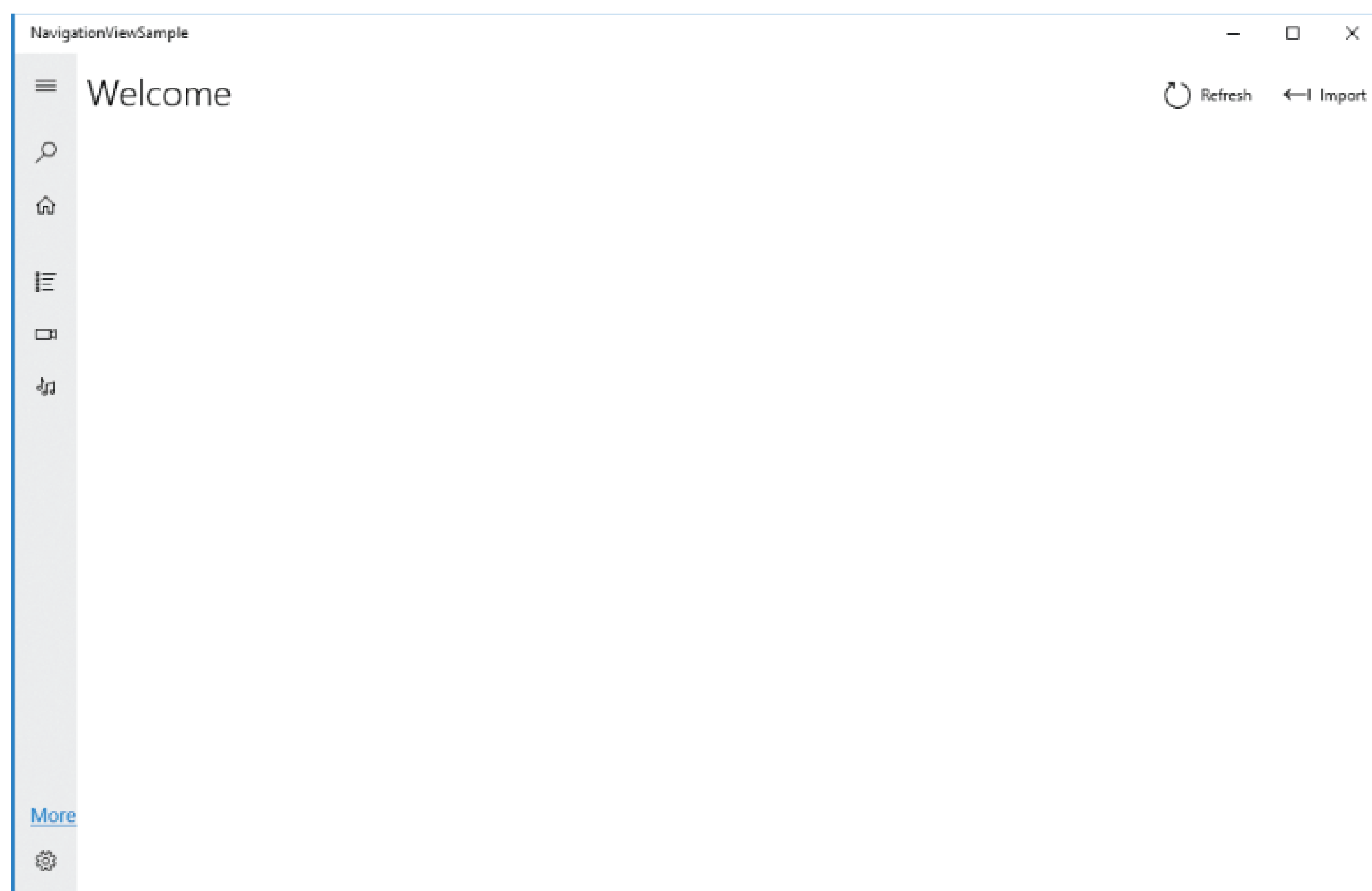



图 33-32

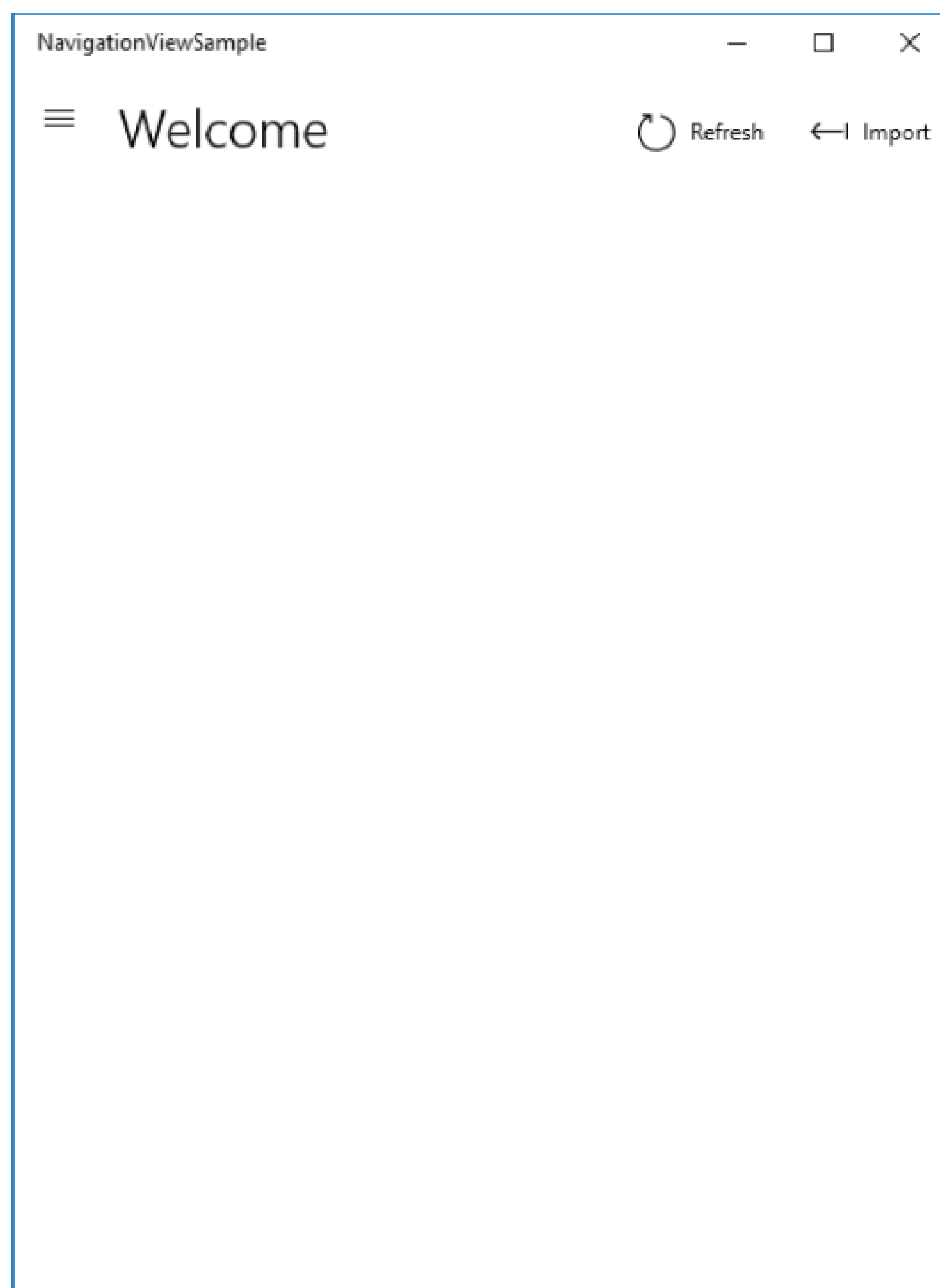



图 33-33

下面讨论 NavigationView 的特性。图 33-34 突出显示了 NavigationView 的不同部分。NavigationView 中定义的第一部分是 MenuItems 列表。这个列表包含 NavigationViewItem 对象。每一项都包含 Icon、Content 和 Tag。可以通过编程方式使用 Tag 来利用这些信息进行导航。对于其中的一些项，使用预定义的图标。用 home 标记的 NavigationViewItem 使用 Unicode 编号为 E10F 的 FontIcon。要分离菜单项，可以使用 NavigationViewItemSeparator。在 NavigationViewItemHeader 中，可以为一组项指定标题内容。注意在窗格处于紧

凑模式时不要剪切该内容。在下面的代码片段中，如果窗格没有完全打开，则会隐藏 NavigationViewItemHeader(代码文件 NavigationViewSample/MainPage.xaml)：

```
<NavigationView x:Name="NavigationView1"
  Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
  <NavigationView.MenuItems>
    <NavigationViewItem Content="Home" Tag="home">
      <NavigationViewItem.Icon>
        <FontIcon Glyph="&#xE10F;"/>
      </NavigationViewItem.Icon>
    </NavigationViewItem>
    <NavigationViewItemSeparator/>
    <NavigationViewItemHeader Content="Main Tools"
      Visibility="{x:Bind NavigationView1.IsPaneOpen, Mode=OneWay}"/>
    <NavigationViewItem Icon="AllApps" Content="Apps" Tag="apps"/>
    <NavigationViewItem Icon="Video" Content="Games" Tag="games"/>
    <NavigationViewItem Icon="Audio" Content="Music" Tag="music"/>
  </NavigationView.MenuItems>

  <!-- ... -->

</NavigationView>
```

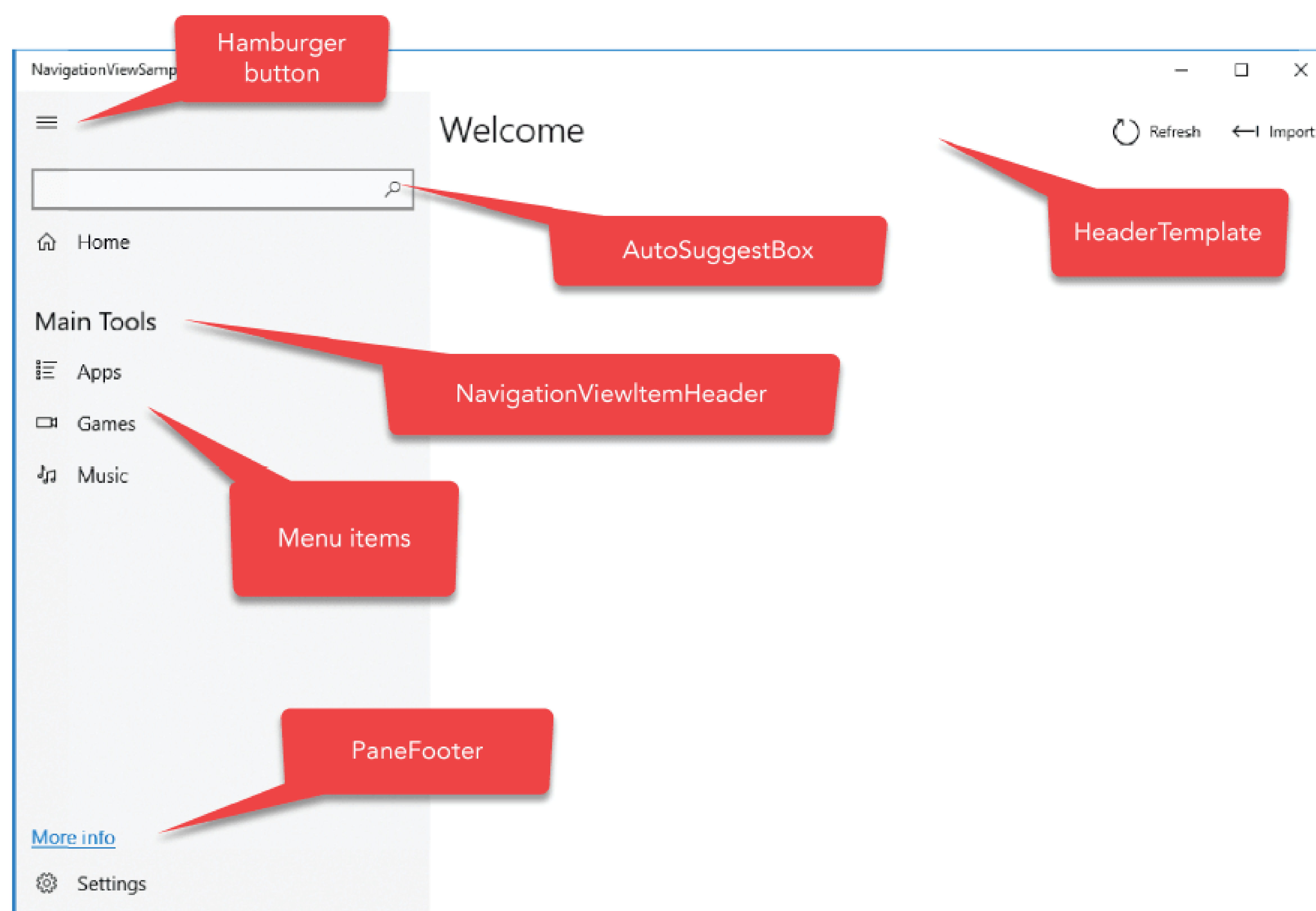



图 33-34

NavigationView 的 AutoSuggestBox 属性允许向导航添加一个 AutoSuggestsBox 控件。这显示在菜单项的顶部。AutoSuggestBox 参见第 36 章(代码文件 NavigationViewSample/MainPage.xaml)：

```
<NavigationView.AutoSuggestBox>
  <AutoSuggestBox x:Name="autoSuggest" QueryIcon="Find"/>
</NavigationView.AutoSuggestBox>
```

使用 HeaderTemplate，可以定制应用程序的顶部。下面的代码片段定义了一个带有 Grid、TextBlock 和 CommandBar 的标题模板(代码文件 NavigationViewSample/MainPage.xaml)：

```
<NavigationView.HeaderTemplate>
  <DataTemplate>
    <Grid Margin="8,8,0,0">
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto"/>
        <ColumnDefinition/>
      </Grid.ColumnDefinitions>
      <TextBlock Style="{StaticResource TitleTextBlockStyle}"
```

```
          FontSize="28"
          VerticalAlignment="Center"
          Text="Welcome"/>
        <CommandBar Grid.Column="1"
          DefaultLabelPosition="Right"
          Background="{ThemeResource SystemControlBackgroundAltHighBrush}">
          <AppBarButton Label="Refresh" Icon="Refresh"/>
          <AppBarButton Label="Import" Icon="Import"/>
        </CommandBar>
      </Grid>
    </DataTemplate>
  </NavigationView.HeaderTemplate>
```

PaneFooter 定义了窗格的下半部分。在页脚下方，默认显示 Settings 的菜单项；这个菜单是默认包含的，由许多应用程序使用(代码文件 NavigationViewSample/MainPage.xaml)：

```
<NavigationView.PaneFooter>
  <HyperlinkButton x:Name="MoreInfoBtn"
    Content="More info"
    Margin="12,0"/>
</NavigationView.PaneFooter>
```

最后，NavigationPane 的内容被 Frame 控件覆盖。此控件用于导航到页面。NavigationPane 围绕页面内容(代码文件 NavigationViewSample/MainPage.xaml)：

```
<Frame x:Name="ContentFrame" Margin="24">
  <Frame.ContentTransitions>
    <TransitionCollection>
      <NavigationThemeTransition/>
    </TransitionCollection>
  </Frame.ContentTransitions>
</Frame>
```

## 33.6 布局

前一节中讨论的 NavigationView 控件是组织用户界面布局的一个重要控件。在许多新的 Windows 10 应用程序中，可以看到这个控件用于主要布局。其他几个控件也定义布局。本节演示了 Variable SizedWrapGrid 在网格中安排自动包装的多个项，RelativePanel 相对于彼此安排各项或相对于父项安排子项，自适应触发器根据窗口的大小重新排列布局。

### 33.6.1 StackPanel

作为其内容，如果要在只能包含一个元素的控件中包含多个元素，最简单的方式就是使用 StackPanel。StackPanel 是一个简单的面板，只能逐个地显示元素。StackPanel 的方向可以是水平或垂直。

在下面的代码片段中，页面包含了一个 StackPanel，其中包含了垂直放置的各个控件。在第一个 ListBoxItem 的列表框中，包含一个横向排列的 StackPanel(代码文件 LayoutSamples/Views/StackPanelPage.xaml)：

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
  <StackPanel Orientation="Vertical">
    <TextBox Text="TextBox" />
    <CheckBox Content="Checkbox" />
    <CheckBox Content="Checkbox" />
    <ListBox>
      <ListBoxItem>
        <StackPanel Orientation="Horizontal">
          <TextBlock Text="One A" />
          <TextBlock Text="One B" />
        </StackPanel>
      </ListBoxItem>
      <ListBoxItem Content="Two" />
    </ListBox>
    <Button Content="Button" />
  </StackPanel>
</Grid>
```

在图 33-35 中，可以看到 StackPanel 垂直显示的子控件。

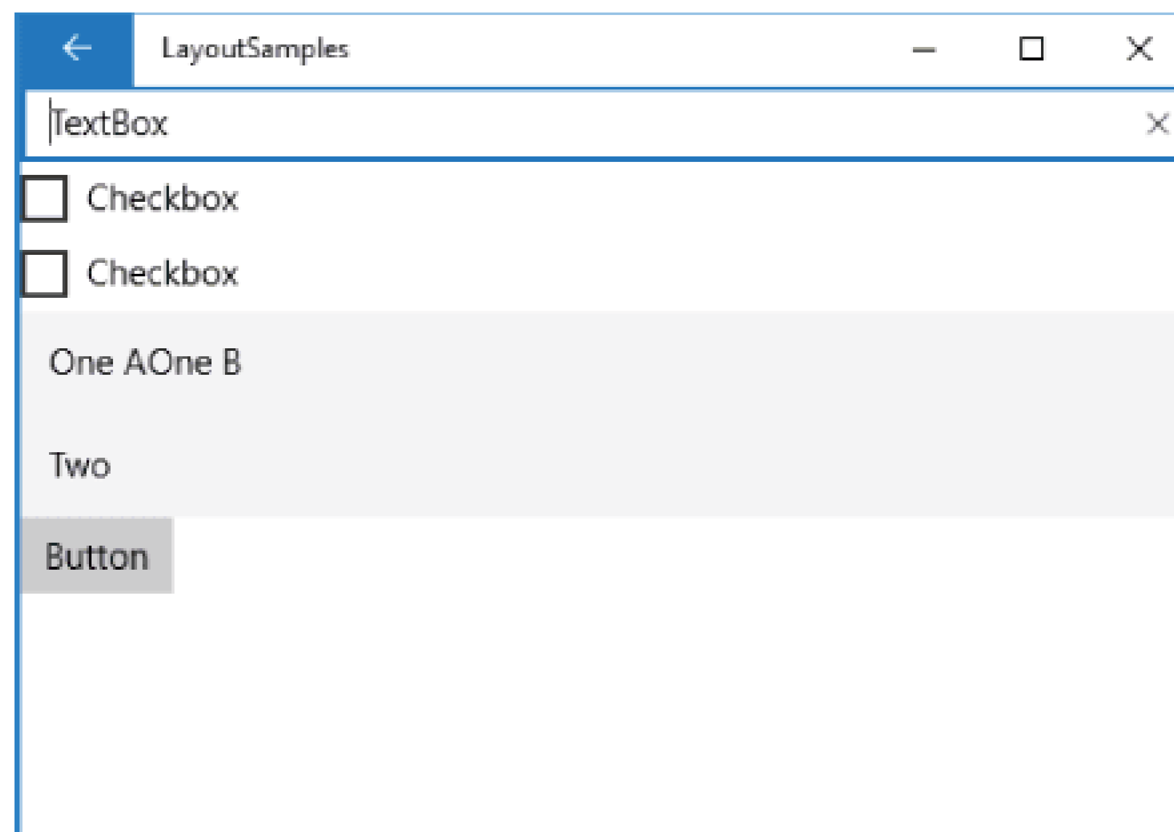



图 33-35

### 33.6.2　Canvas

Canvas 是一个允许显式指定控件位置的面板。它定义了相关的 Left、Right、Top 和 Bottom 属性，这些属性可以由子元素在面板中定位时使用(代码文件 LayoutSamples/Views/CanvasPage.xaml)。

图 33-36 显示了 Canvas 面板的结果，其中定位了子元素 TextBlock、TextBox 和 Button。

```
<Canvas Background="LightBlue">
  <TextBlock Canvas.Top="30" Canvas.Left="20">Enter here:</TextBlock>
  <TextBox Canvas.Top="30" Canvas.Left="120" Width="100" />
  <Button Canvas.Top="70" Canvas.Left="120" Content="Click Me!" Padding="4" />
</Canvas>
```

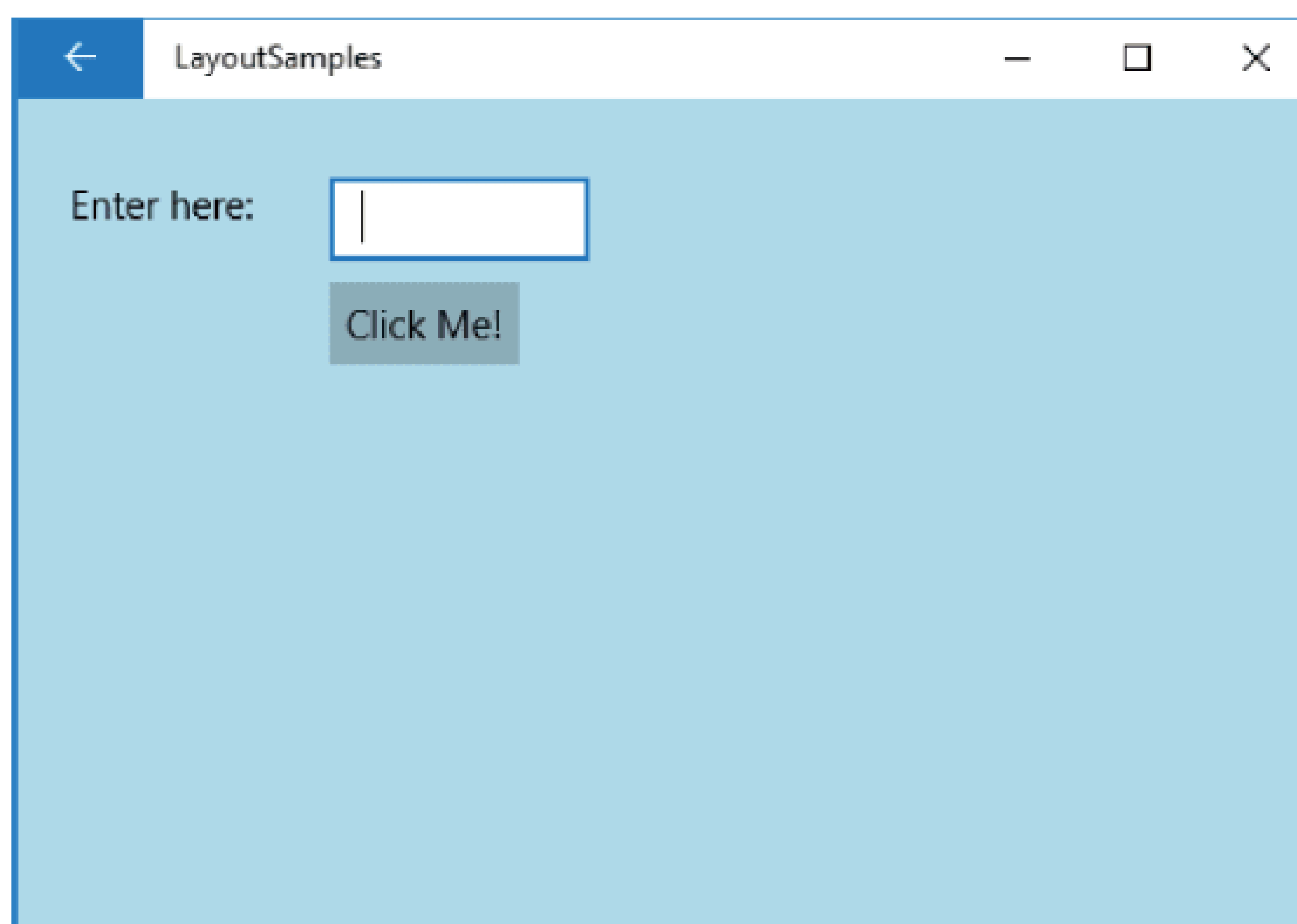



图 33-36

**注意：**

Canvas 控件最适合用于图形元素的布局，例如第 35 章介绍的 Shape 控件。

### 33.6.3　Grid

Grid 是一个重要的面板。使用 Grid，可以在行和列中排列控件。对于每一列，可以指定一个 ColumnDefinition；对于每一行，可以指定一个 RowDefinition。下面的示例代码显示两列和三行。在每一列和每一行中，都可以指定宽度或高度。ColumnDefinition 有一个 Width 依赖属性，RowDefinition 有一个 Height 依赖属性。可以以设备独立的像素为单位定义高度和宽度，或者把它们设置为 Auto，根据内容来确定其大小。Grid 还允许使用“星型大小”，即根据具体情况指定大小，即根据可用的空间以及与其他行和列的相对位置计算行和列的空间。在为列提供可用空间时，可以将 Width 属性设置为“*”。要使某一列的空间是另一列的两倍，应指定“2*”。下面的示例代码定义了两列和三行，列使用“星型大小”，第一行的大小固定，第二行和第三行再次使用“星型大小”。在计算高度时，可用空间需要减去第一行的 200 像素，剩余的区域在第二行和第三行中按比例 1.5:1 来分配。

这个 Grid 包含几个 Rectangle 控件，它们用不同的颜色使单元格的尺寸可见。因为这些控件的父控件是 Grid，所以可以设置附加属性 Column、ColumnSpan、Row 和 RowSpan(代码文件 LayoutSamples/Views/GridPage.xaml)。

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
  <Grid.ColumnDefinitions>
    <ColumnDefinition />
    <ColumnDefinition />
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition Height="200" />
    <RowDefinition Height="1.5*" />
    <RowDefinition Height="*" />
  </Grid.RowDefinitions>
  <Rectangle Fill="Blue" />
  <Rectangle Grid.Row="0" Grid.Column="1" Fill="Red" />
  <Rectangle Grid.Row="1" Grid.Column="0" Grid.ColumnSpan="2" Fill="Green" />
```

```
  <Rectangle Grid.Row="2" Grid.Column="0" Grid.ColumnSpan="2" Fill="Yellow" />
</Grid>
```

在 Grid 中排列控件的结果如图 33-37 所示。

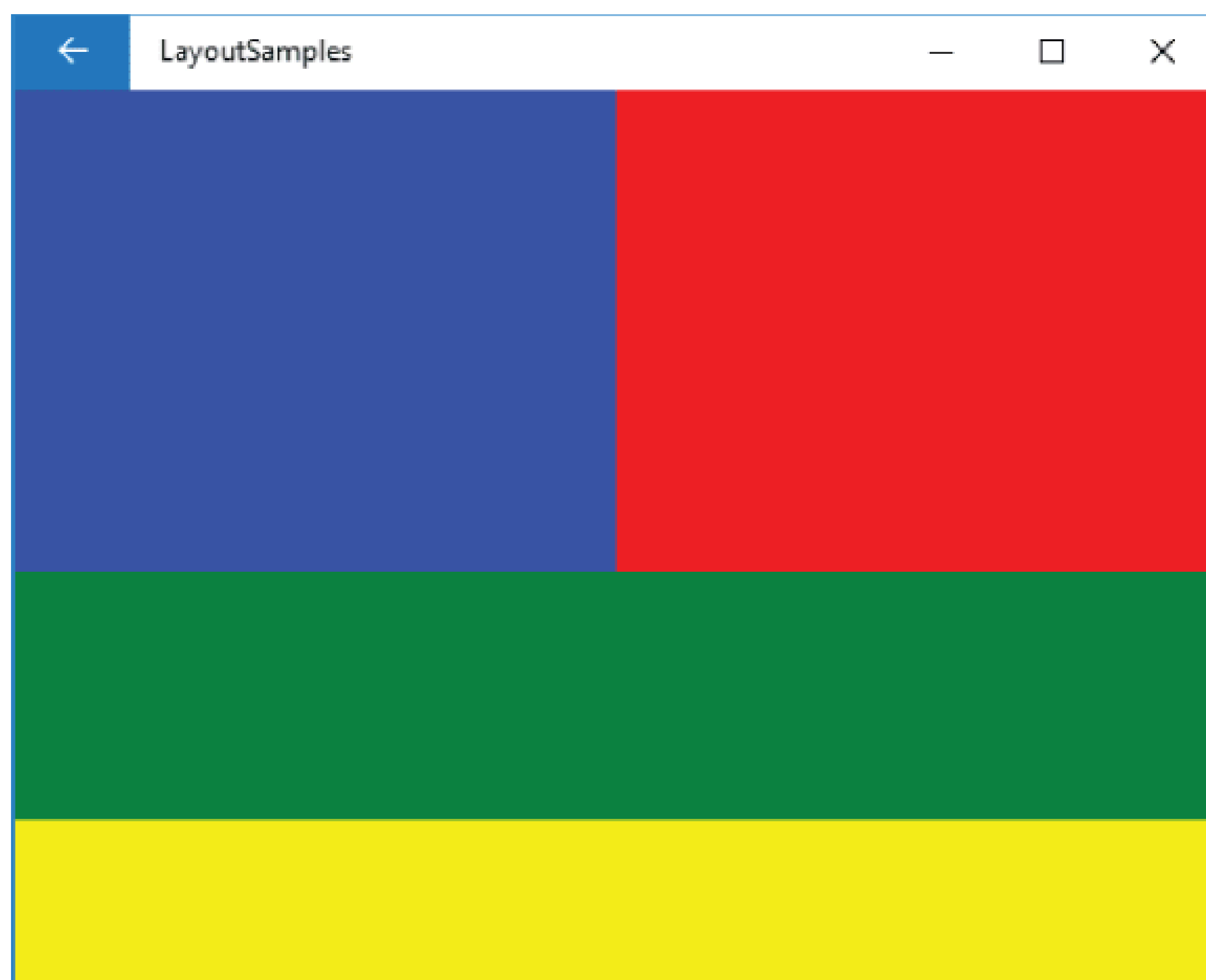



图 33-37

### 33.6.4 VariableSizedWrapGrid

VariableSizedWrapGrid 是一个包装网格，如果网格可用的大小不够大，它会自动换到下一行或列。这个表格的第二个特征是允许项放在多行或多列中，这就是为什么它称为可变的原因。

下面的代码片段创建一个 VariableSizedWrappedGrid，其方向是 Horizontal，行中最多有 20 项，行和列的大小是 50(代码文件 LayoutSamples/Views/VariableSizedWrapGridSample.xaml)：

```
<VariableSizedWrapGrid x:Name="grid1" MaximumRowsOrColumns="20" ItemHeight="50"
  ItemWidth="50" Orientation="Horizontal" />
```

VariableSizedWrapGrid 填充了 30 个随机大小和颜色的 Rectangle 和 TextBlock 元素。根据大小，可以在网格内使用 1 到 3 行或列。项的大小使用附加属性 VariableSizedWrapGrid.ColumnSpan 和 VariableSizedWrap-Grid.RowSpan 设置(代码文件 LayoutSamples/Views/VariableSizedWrapGridSample. xaml.cs)：

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
  base.OnNavigatedTo(e);
  Random r = new Random();
  Grid[] items =
    Enumerable.Range(0, 30).Select(i =>
    {
      byte[] colorBytes = new byte[3];
      r.NextBytes(colorBytes);
      var rect = new Rectangle
      {
        Height = r.Next(40, 150),
        Width = r.Next(40, 150),
        Fill = new SolidColorBrush(new Color
        {
          R = colorBytes[0],
          G = colorBytes[1],
          B = colorBytes[2],
          A = 255
        })
      };

      var textBlock = new TextBlock
      {
        Text = (i + 1).ToString(),
        HorizontalAlignment = HorizontalAlignment.Center,
        VerticalAlignment = VerticalAlignment.Center
      };
```

```
  var grid = new Grid();
  grid.Children.Add(rect);
  grid.Children.Add(textBlock);
  return grid;
}).ToArray();

foreach (var item in items)
{
  grid1.Children.Add(item);
  Rectangle rect = item.Children.First() as Rectangle;
  if (rect.Width > 50)
  {
    int columnSpan = ((int)rect.Width / 50) + 1;
    VariableSizedWrapGrid.SetColumnSpan(item, columnSpan);
    int rowSpan = ((int)rect.Height / 50) + 1;
    VariableSizedWrapGrid.SetRowSpan(item, rowSpan);
  }
}
```

运行应用程序时，可以看到矩形，它们占用了不同的窗口，如图 33-38 和图 33-39 所示。




图 33-38

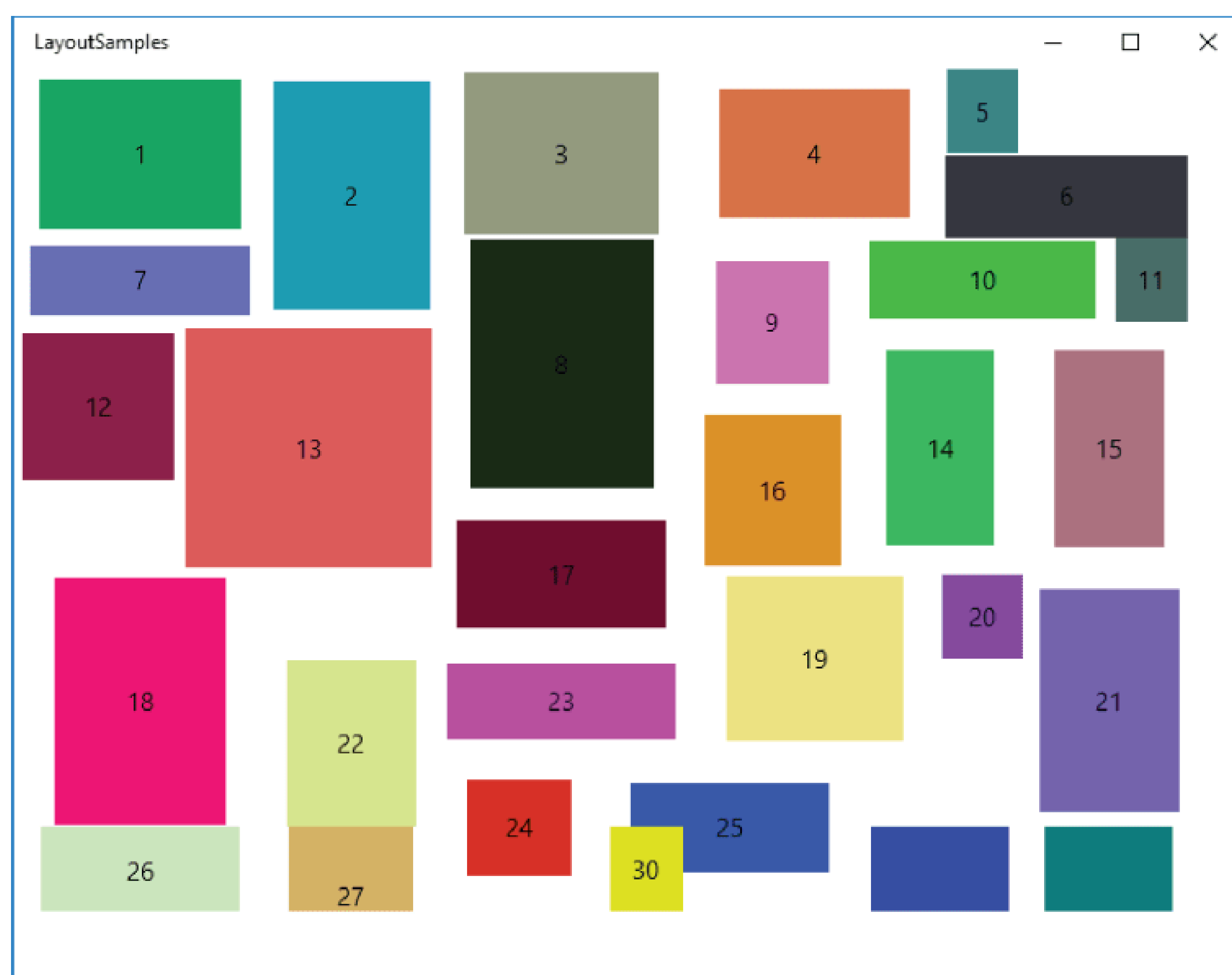



图 33-39

### 33.6.5　RelativePanel

RelativePanel 是 UWP 的一个新面板，允许一个元素相对于另一个元素定位。如果使用的 Grid 控件定义了行和列，且需要插入一行，就必须修改插入行下面的所有元素。原因是所有行和列都按数字索引。使用 RelativePanel 就没有这个问题，它允许根据元素的相对关系放置它们。

> **注意：**
> 与 RelativePanel 相比，Grid 控件仍然有它的自动、星形和固定大小的优势。

下面的代码片段在 RelativePanel 内对齐数个 TextBlock 和 TextBox 控件、一个按钮和一个矩形。TextBox 元素定位在相应 TextBlock 元素的右边；按钮相对于面板的底部定位，矩形与第一个 TextBlock 的顶部对齐，与第一个 TextBox 的右边对齐(代码文件 LayoutSamples/Views/RelativePanelPage.xaml)：

```
<RelativePanel>
  <TextBlock x:Name="FirstNameLabel" Text="First Name" Margin="8" />
  <TextBox x:Name="FirstNameText" RelativePanel.RightOf="FirstNameLabel"
    Margin="8" Width="150" />
  <TextBlock x:Name="LastNameLabel" Text="Last Name"
    RelativePanel.Below="FirstNameLabel" Margin="8" />
  <TextBox x:Name="LastNameText" RelativePanel.RightOf="LastNameLabel"
    Margin="8" RelativePanel.Below="FirstNameText" Width="150" />
  <Button Content="Save" RelativePanel.AlignHorizontalCenterWith="LastNameText"
    RelativePanel.AlignBottomWithPanel="True" Margin="8" />
  <Rectangle x:Name="Image" Fill="Violet" Width="150" Height="250"
    RelativePanel.AlignTopWith="FirstNameLabel"
    RelativePanel.RightOf="FirstNameText" Margin="8" />
</RelativePanel>
```

图 33-40 显示了运行应用程序时对齐控件。

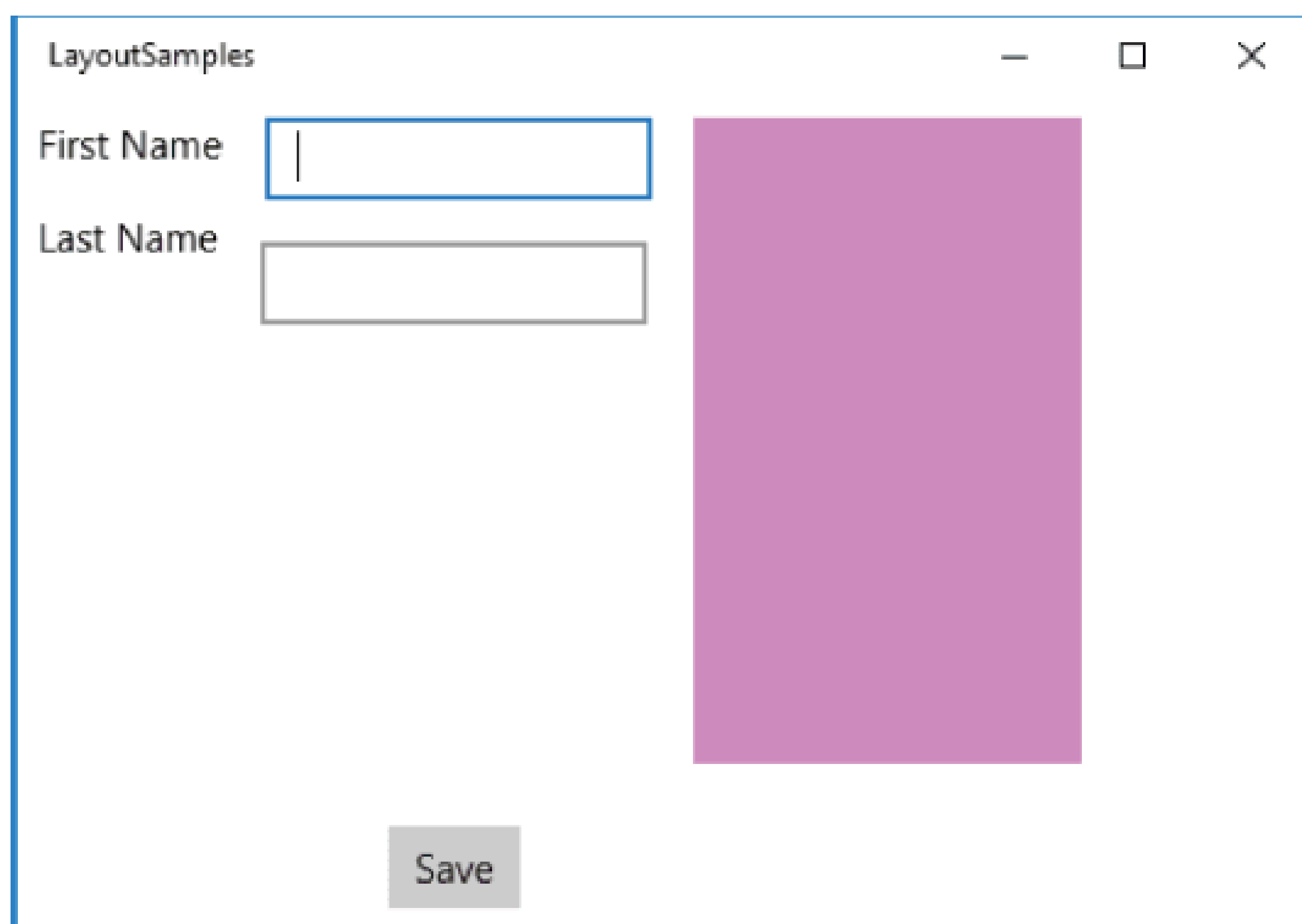



图 33-40

### 33.6.6　自适应触发器

RelativePanel 是用于对齐的一个好控件。但是，为了支持多个屏幕大小，根据屏幕大小重新排列控件，可以使用自适应触发器与 RelativePanel 控件。例如，在小屏幕上，TextBox 控件应该安排在 TextBlock 控件的下方，但在大的屏幕上，TextBox 控件应该在 TextBlock 控件的右边。

在以下代码中，之前的 RelativePanel 改为删除 RelativePanel 中不应用于所有屏幕尺寸的所有附加属性，添加一个可选的图片(代码文件 LayoutSamples/Views/AdaptiveRelativePanelPage.xaml)：

```
<RelativePanel ScrollViewer.VerticalScrollBarVisibility="Auto" Margin="16">
  <TextBlock x:Name="FirstNameLabel" Text="First Name" Margin="8" />
  <TextBox x:Name="FirstNameText" Margin="8" Width="150" />
  <TextBlock x:Name="LastNameLabel" Text="Last Name" Margin="8" />
  <TextBox x:Name="LastNameText" Margin="8" Width="150" />
  <Button Content="Save" RelativePanel.AlignBottomWithPanel="True"
    Margin="8" />
```

```
    <Rectangle x:Name="Image" Fill="Violet" Width="150" Height="250"
      Margin="8" />
    <Rectangle x:Name="OptionalImage" RelativePanel.AlignRightWithPanel="True"
      Fill="Red" Width="350" Height="350" Margin="8" />
  </RelativePanel>
```

使用自适应触发器(当启动触发器时，可以使用自适应触发器设置 MinWindowWidth)，设置不同的属性值，根据应用程序可用的空间安排元素。随着屏幕尺寸越来越小，这个应用程序所需的宽度也会变小。向下移动元素，而不是向旁边移动，可以减少所需的宽度。另外，用户可以向下滚动。对于最小的窗口宽度，可选图像设置为收缩(代码文件 LayoutSamples/Views/AdaptiveRelativePanelPage.xaml)：

```
<VisualStateManager.VisualStateGroups>
  <VisualStateGroup>
    <VisualState x:Name="WideState">
      <VisualState.StateTriggers>
        <AdaptiveTrigger MinWindowWidth="1024" />
      </VisualState.StateTriggers>
      <VisualState.Setters>
        <Setter Target="FirstNameText.(RelativePanel.RightOf)"
          Value="FirstNameLabel" />
        <Setter Target="LastNameLabel.(RelativePanel.Below)"
          Value="FirstNameLabel" />
        <Setter Target="LastNameText.(RelativePanel.Below)"
          Value="FirstNameText" />
        <Setter Target="LastNameText.(RelativePanel.RightOf)"
          Value="LastNameLabel" />
        <Setter Target="Image.(RelativePanel.AlignTopWith)"
          Value="FirstNameLabel" />
        <Setter Target="Image.(RelativePanel.RightOf)" Value="FirstNameText" />
      </VisualState.Setters>
    </VisualState>
    <VisualState x:Name="MediumState">
      <VisualState.StateTriggers>
        <AdaptiveTrigger MinWindowWidth="720" />
      </VisualState.StateTriggers>
      <VisualState.Setters>
        <Setter Target="FirstNameText.(RelativePanel.RightOf)"
          Value="FirstNameLabel" />
        <Setter Target="LastNameLabel.(RelativePanel.Below)"
          Value="FirstNameLabel" />
        <Setter Target="LastNameText.(RelativePanel.Below)"
          Value="FirstNameText" />
        <Setter Target="LastNameText.(RelativePanel.RightOf)"
          Value="LastNameLabel" />
        <Setter Target="Image.(RelativePanel.Below)" Value="LastNameText" />
        <Setter Target="Image.(RelativePanel.AlignHorizontalCenterWith)"
          Value="LastNameText" />
      </VisualState.Setters>
    </VisualState>
    <VisualState x:Name="NarrowState">
      <VisualState.StateTriggers>
        <AdaptiveTrigger MinWindowWidth="320" />
      </VisualState.StateTriggers>
      <VisualState.Setters>
        <Setter Target="FirstNameText.(RelativePanel.Below)"
          Value="FirstNameLabel" />
        <Setter Target="LastNameLabel.(RelativePanel.Below)"
          Value="FirstNameText" />
        <Setter Target="LastNameText.(RelativePanel.Below)"
          Value="LastNameLabel" />
        <Setter Target="Image.(RelativePanel.Below)" Value="LastNameText" />
        <Setter Target="OptionalImage.Visibility" Value="Collapsed" />
      </VisualState.Setters>
    </VisualState>
  </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

通过 ApplicationView 类设置 SetPreferredMinSize，可以建立应用程序所需的最小窗口宽度(代码文件

LayoutSamples/App.xaml.cs)：

```
protected override void OnLaunched(LaunchActivatedEventArgs e)
{
  ApplicationView.GetForCurrentView().SetPreferredMinSize(
  new Size { Width = 320, Height = 300 });
  //...
}
```

运行应用程序时，可以看到最小宽度的布局安排(见图 33-41)、中等宽度的布局安排(见图 33-42)和最大宽度的布局安排(见图 33-43)。

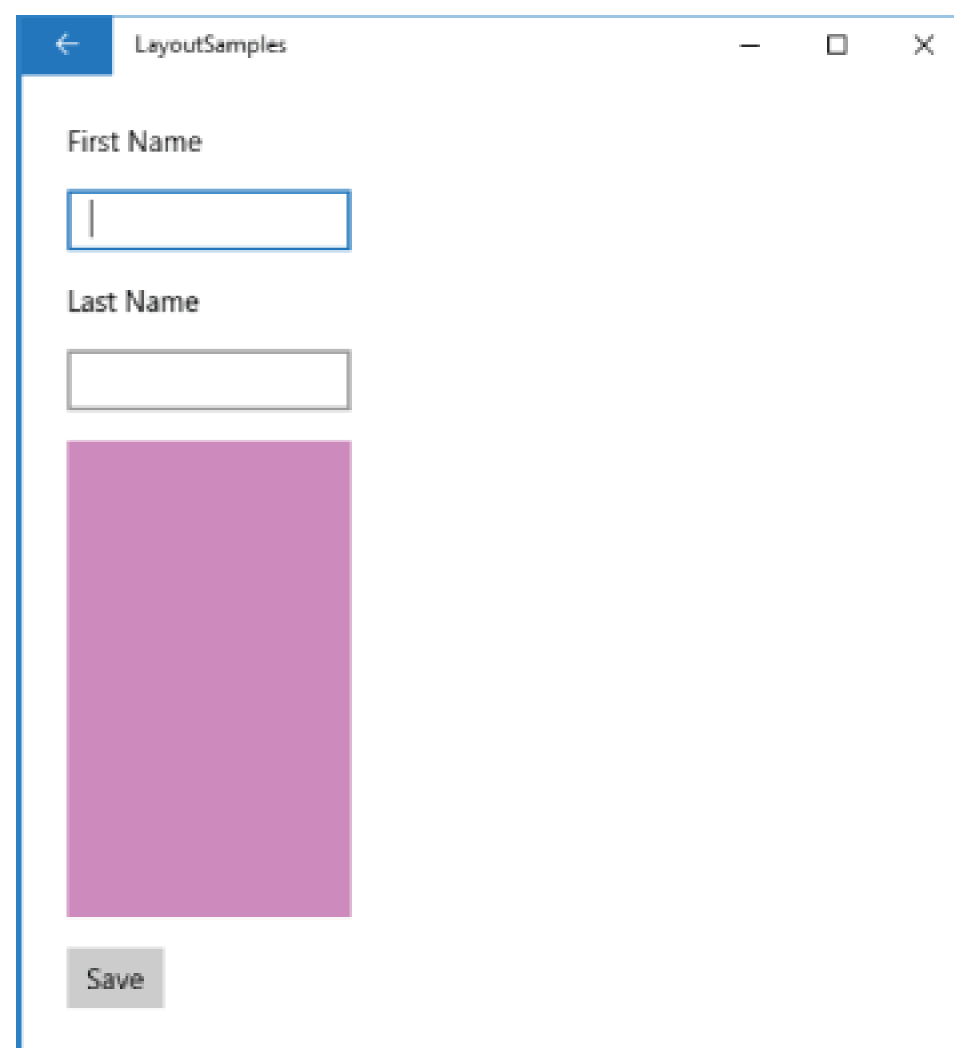



图 33-41

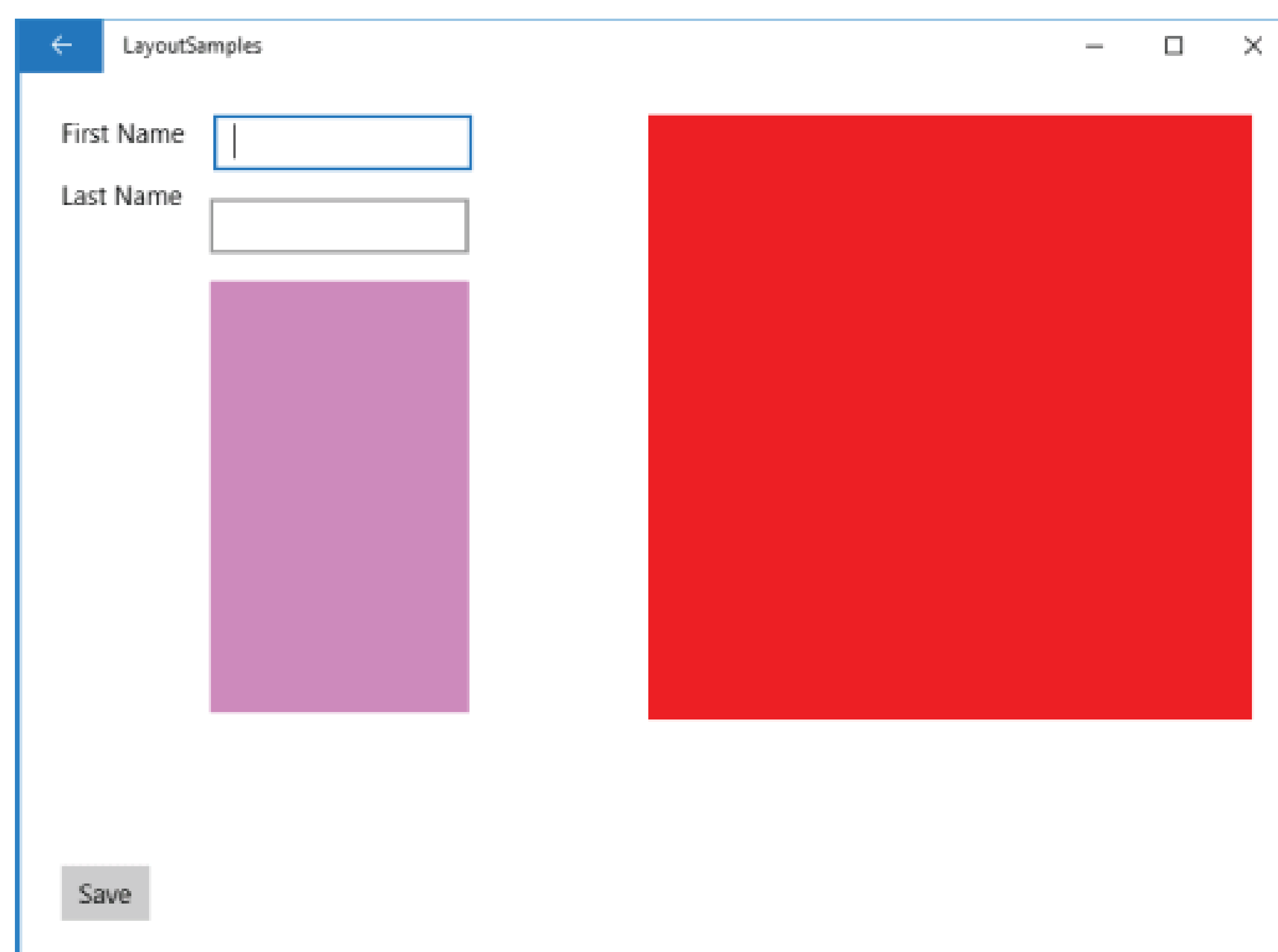



图 33-42

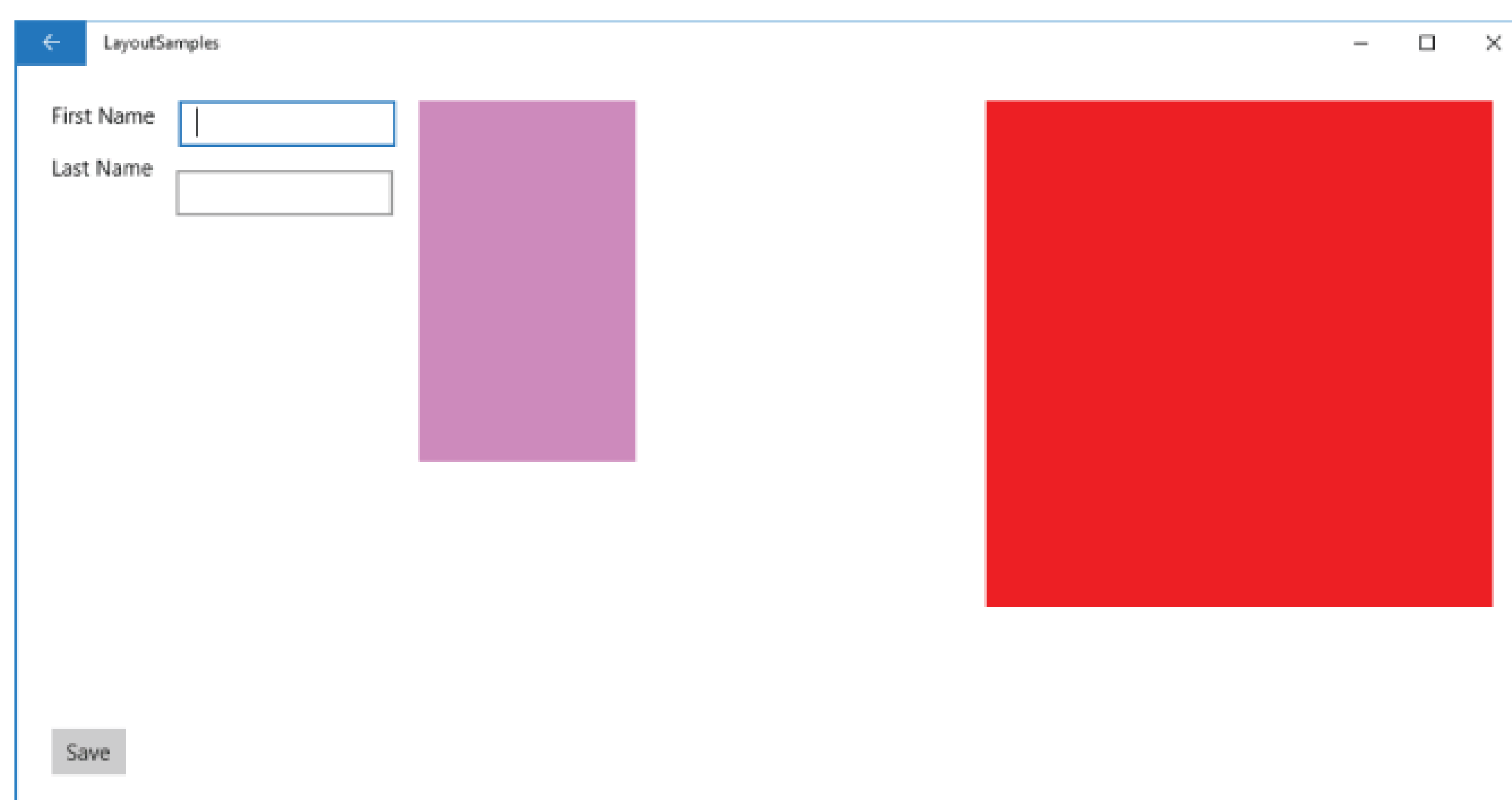



图 33-43

### 33.6.7　XAML 视图

自适应触发器可以帮助支持很多不同的窗口大小，支持应用程序的布局，以便在 Xbox、HoloLens 和不同分辨率的桌面上运行。如果应用程序的用户界面应该有比使用 RelativePanel 更多的差异，最好的选择是使用不同的 XAML 视图。XAML 视图只包含 XAML 代码，并使用与相应页面相同的代码隐藏文件。可以为每个设备系列创建同一个页面的不同 XAML 视图。

通过创建一个文件夹 DeviceFamily-Mobile，可以为移动设备定义 XAML 视图。设备专用的文件夹总是以 DeviceFamily 名称开头。支持的其他设备系列有 Team、Desktop 和 IoT。可以使用这个设备系列的名字作为后缀，指定相应设备系列的 XAML 视图。使用 XAML View Visual Studio 项模板创建一个 XAML 视图。这个模板创建 XAML 代码，但没有代码隐藏文件。这个视图需要与应该更换视图的页面同名。

除了为移动 XAML 视图创建另一个文件夹之外，还可以在页面所在的文件夹中创建视图，但视图文件使用 DeviceFamily-Mobile 命名。

### 33.6.8　延迟加载

为了使 UI 更快，可以把控件的创建延迟到需要它们时再创建。在小型设备上，可能根本不需要一些控件，但如果系统使用较大的屏幕，也比较快，就需要这些控件。在 XAML 应用程序的先前版本中，添加到 XAML 代码中的元素也被实例化。Windows 10 不再是这种情况，而可以把控件的加载延迟到需要它们时加载。

可以使用延迟加载和自适应触发器，只在稍后的时间加载一些控件。一个样本场景是，用户可以把小窗口调整得更大。在小窗口中，有些控件不应该是可见的，但它们应该在更大的窗口中可见。延迟加载可能有用的另一个场景是，布局的某些部分可能需要更多时间来加载。不是让用户等待，直到显示出完整加载的布局，而可以使用延迟加载。

要使用延迟加载，需要给控件添加 x: Load 特性(其值为 False)，如下面带有 Grid 控件的代码片段所示。这个控件也需要分配一个名字(代码文件 LayoutSamples/Views/DelayLoadingPage.xaml):

```
<Grid x:Load="False" x:Name="deferGrid">
  <Grid.ColumnDefinitions>
    <ColumnDefinition />
    <ColumnDefinition />
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition />
    <RowDefinition />
  </Grid.RowDefinitions>
  <Rectangle Fill="Red" Grid.Row="0" Grid.Column="0" />
  <Rectangle Fill="Green" Grid.Row="0" Grid.Column="1" />
  <Rectangle Fill="Blue" Grid.Row="1" Grid.Column="0" />
  <Rectangle Fill="Yellow" Grid.Row="1" Grid.Column="1" />
</Grid>
```

为了使这个延迟的控件可见，只需要调用 FindName 方法访问控件的标识符。这不仅使控件可见，而且会在控件可见前加载控件的 XAML 树(代码文件 LayoutSamples/Views/DelayLoadingPage. xaml.cs):

```
private void OnDeferLoad(object sender, RoutedEventArgs e)
{
  FindName(nameof(deferGrid));
}
```

**注意：**

*x:Load 特性是构建版本 15063 中新增的。在构建版本 15063 之前，可以使用 x:DeferLoadingStrategy 特性。x:Load 的优点是元素也可以在加载后卸载。*

运行应用程序时，可以用 Live Visual Tree 窗口验证，包含 deferGrid 元素的树不可用(见图 33-44)，但在调用 FindName 方法找到 deferGrid 元素后，deferGrid 元素就添加到树中(参见图 33-45)。

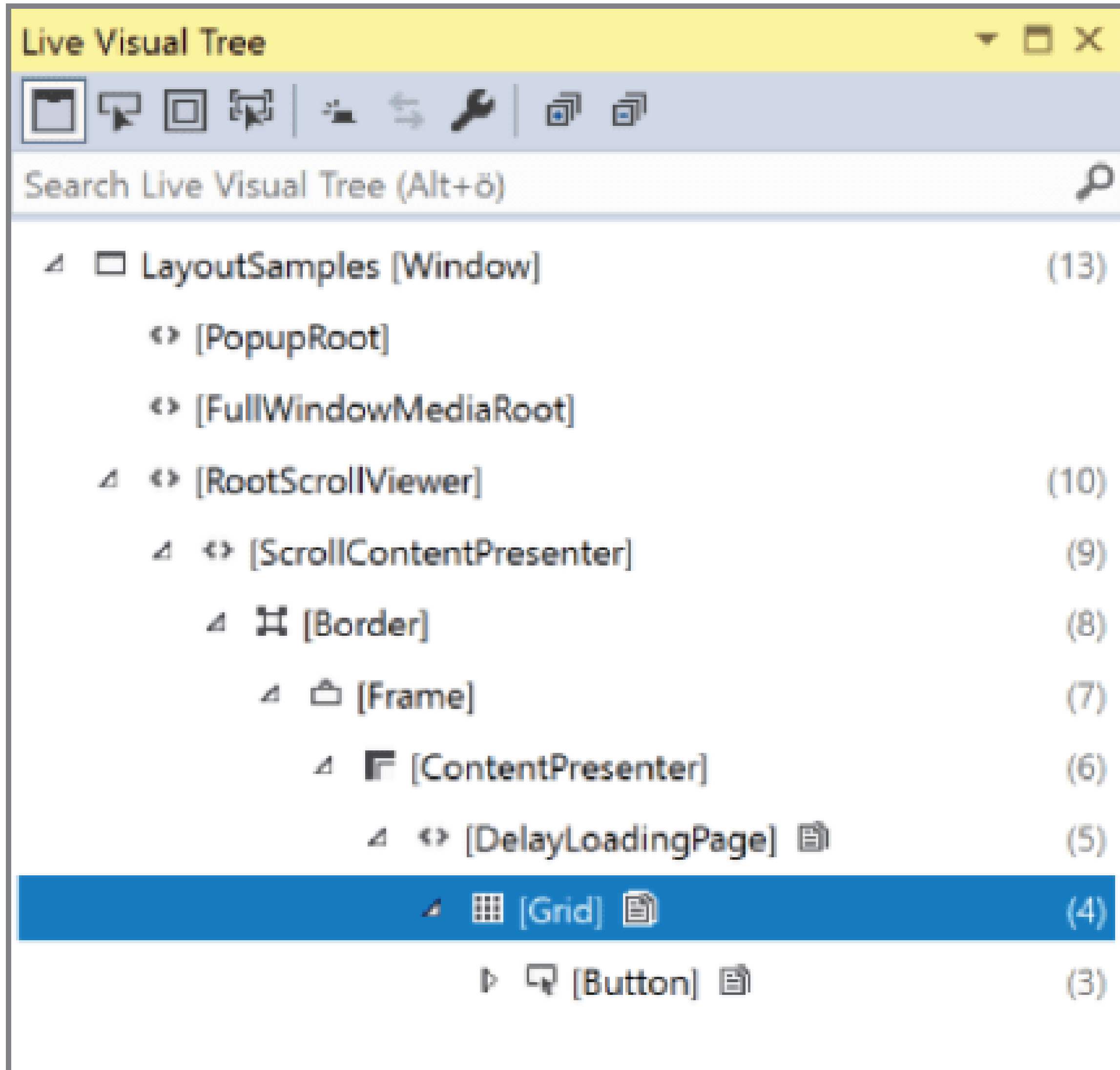



图 33-44

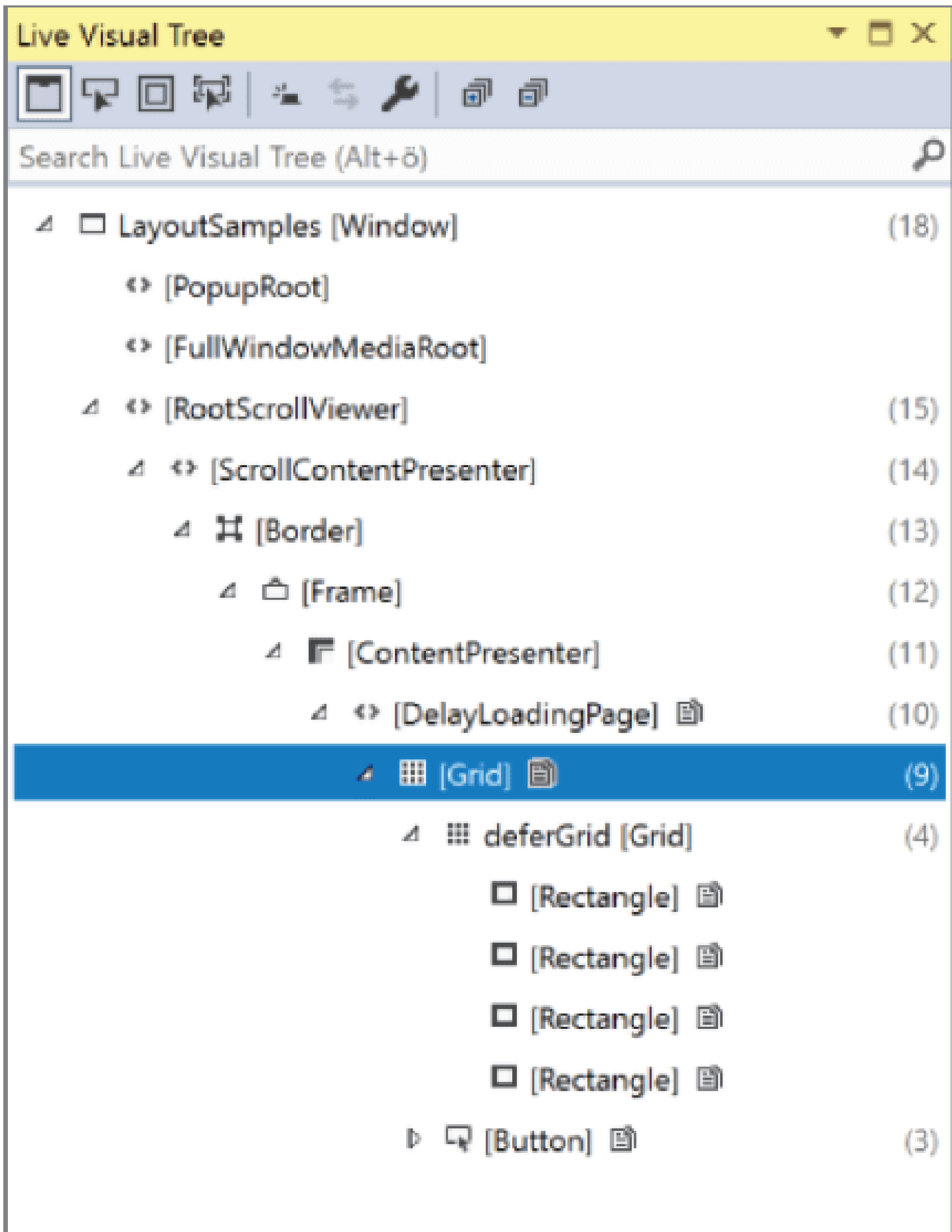



图 33-45

**注意：**

特性 x:Load 有大约 600 字节的开销，所以应该只在需要隐藏的元素上使用它。如果在容器元素上使用此特性，就只需要向应用该特性的元素支付一次开销。

## 33.7 小结

本章介绍了 Windows 应用程序编程的许多不同方面，了解了 XAML 的基础，以及它如何使用附加属性和标记扩展来扩展 XML。学习了如何使用条件 XAML 处理不同 Windows 10 版本中的 XAML 差异。

本章讨论了如何处理不同的屏幕大小、使用不同面板布置控件的选项，以及不同控件的类别和特性。

下一章将继续介绍基于 XAML 的应用程序、MVVM 模式、命令和创建可共享的视图模型。

# 第34章

# 模式和 XAML 应用程序

**本章要点**

- 共享代码
- 创建模型
- 创建存储库
- 创建视图模型
- 页面之间的导航
- 自适应用户界面
- 使用事件聚合器
- 带视图模型的列表项

**本章源代码下载地址(wrox.com)：**

打开 www.wrox.com 的 Download Code 选项卡可下载本章源代码。源代码也可以在 Patterns 和 PatternsXamarinShared 目录的 https://github.com/ProfessionalCSharp/ProfessionalCSharp7 中找到。库在第 34 章和第 37 章之间共享。

本章代码包含项目 BooksApp。

## 34.1 使用 MVVM 的原因

技术和框架一直在改变。我用 ASP.NET Web Forms 创建了公司网站的第 1 版(http://www.cninnovation.com)。在 ASP.NET MVC 出现时，我试着把网站的功能迁移到 MVC。进度比预期的要快得多。一天之内就把完整的网站改为 MVC。该网站使用 SQL Server，集成了 RSS 提要，显示了培训和图书。关于培训和图书的信息来自 SQL Server 数据库。可以快速迁移到 ASP.NET MVC，只是因为我从一开始就分离了关注点，为数据访问和业务逻辑创建了独立的层。有了 ASP.NET Web Forms，可以在 ASPX 页面中直接使用数据源和数据控件。分离数据访问和业务逻辑，一开始花了更多的时间，但它变成一个巨大的优势，由于它允许单元测试和重用。因为以这样的方式进行分离，所以迁移到另一个技术真是太容易了。目前这个站点使用 ASP.NET Core 运行。

对于 Windows 应用程序，技术也变得很快。多年来，Windows Forms 技术包装了本地 Windows 控件，来创建桌面应用程序。之后出现了 Windows Presentation Foundation(WPF)，在其中用户界面使用 eXtensible Application

Markup Language (XAML)定义。Silverlight 为在浏览器中运行的、基于 XAML 的应用程序提供了一个轻量级的框架。Windows Store 应用程序随着 Windows 8 而出现，在 Windows 8.1 中改为通用 Windows 应用程序，运行在个人电脑和 Windows Phone 上。在 Windows 8.1 和 Visual Studio 2013 中，创建了三个带有共享代码的项目，同时支持个人电脑和手机。接着又变成 Visual Studio 2015、Windows 10、通用 Windows 平台(UWP)。一个项目可以支持个人电脑、手机、Xbox One、Windows IoT、带有 Surface Hub 的大屏幕，甚至 Microsoft 的 HoloLens。

一个支持所有 Windows 10 平台的项目可能不满足需求。可以编写一个仅支持 Windows 10 的程序吗？一些客户可能仍在运行 Windows 7。在这种情况下，应使用 WPF，但它不支持手机和其他 Windows 10 设备，如 HoloLens 和 Xbox。如何支持 Android 和 iOS 呢？在这里，可以使用 Xamarin 创建 C#和.NET 代码，但它是不同的。

目标应该是重用尽可能多的代码，支持所需的平台，很容易从一种技术切换到另一种。这些目标(在许多组织中，管理和开发部门加入 DevOps，会很快给用户带来新的功能，修复缺陷)要求自动化测试。单元测试是必需的，应用程序体系结构需要支持它。

> **注意：**
> 单元测试参见第 28 章。

有了基于 XAML 的应用程序，Model-View-ViewModel(MVVM)设计模式便于分离视图和功能。该设计模式是由 Expression Blend 团队的 John Gossman 发明，能更好地适应 XAML，改进了 Model-View-Controller (MVC)和 Model-View-Presenter(MVP)模式，因为它使用了 XAML 的首要功能：数据绑定。

有了基于 XAML 的应用程序，XAML 文件和代码隐藏文件是紧密耦合的。这很难重用代码隐藏文件，单元测试也很难做到。为了解决这个问题，人们提出了 MVVM 模式，它允许更好地分离用户界面和代码。

原则上，MVVM 模式并不难理解。然而，基于 MVVM 模式创建应用程序时，需要注意更多的需求：几个模式会发挥作用，使应用程序工作起来，使重用成为可能，包括依赖注入机制独立于视图模型的实现和视图模型之间的通信。

本章介绍这些内容，有了这些信息，不仅可以给 Windows 应用程序和桌面应用程序使用相同的代码，还可以在 Xamarin 的帮助下把它用于 iOS 和 Android。本章给出一个示例应用程序，其中包括了所有不同的方面和模式，实现很好的分离，支持不同的技术。

## 34.2 定义 MVVM 模式

首先看看 MVVM 模式的起源之一：MVC 设计模式。Model-View-Controller (MVC)模式分离了模型、视图和控制器(见图 34-1)。模型定义视图中显示的数据，以及改变和操纵数据的业务规则。控制器是模型和视图之间的管理器，它会更新模型，给视图发送要显示的数据。当用户请求传入时，控制器就采取行动，使用模型，更新视图。

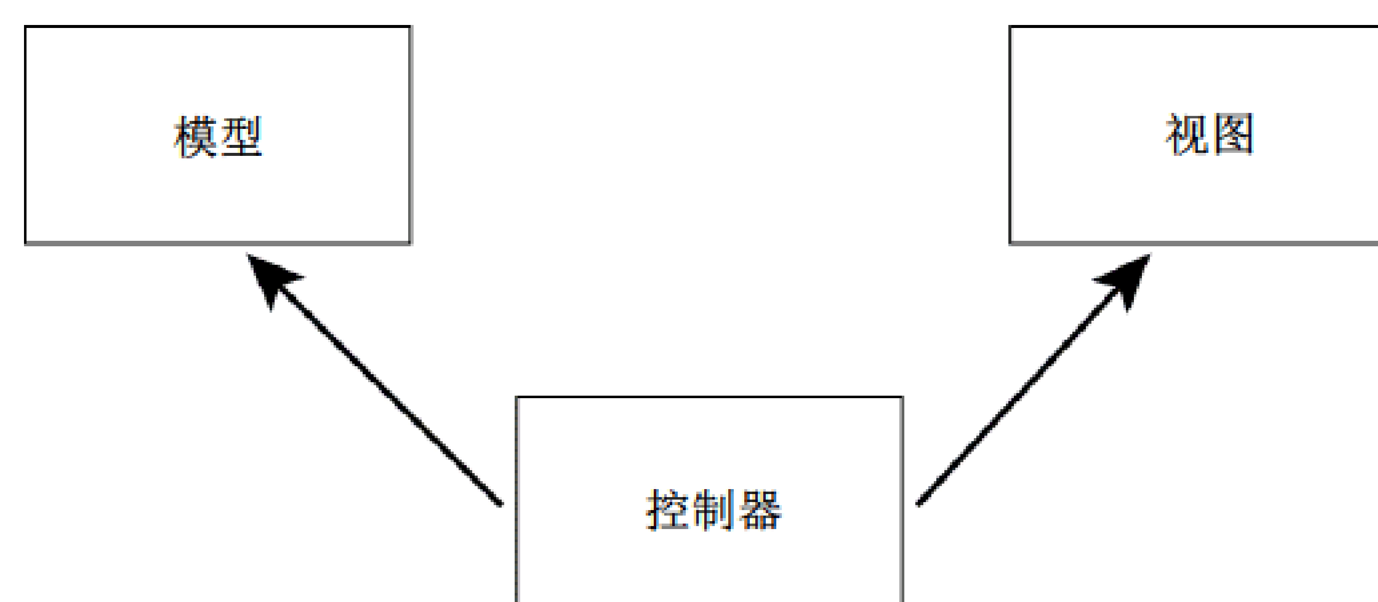



图 34-1

> **注意：**
> MVC 模式大量用于 ASP.NET MVC，参见第 31 章。

通过 Model-View-Presenter(MVP)模式(见图 34-2)，用户与视图交互操作。显示程序包含视图的所有业务逻

辑。显示程序可以使用一个视图的接口作为协定，从视图中解除耦合。这样就很容易改变单元测试的视图实现。在 MVP 中，视图和模型是完全相互隔离的。

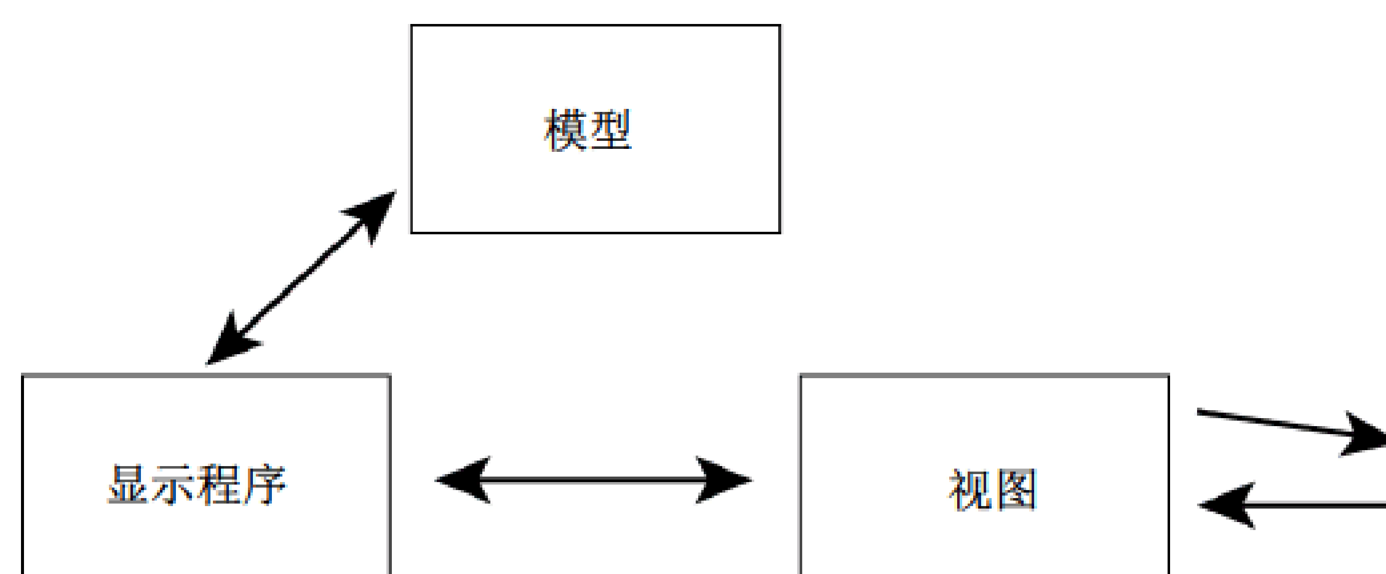



图 34-2

基于 XAML 的应用程序使用的主要模式是 Model-View-ViewModel(MVVM)(见图 34-3)。这种模式利用数据绑定功能与 XAML。通过 MVVM，用户与视图交互。视图使用数据绑定来访问视图模型的信息，并在绑定到视图上的视图模型中调用命令。视图模型没有对视图的直接依赖项。视图模型本身使用模型来访问数据，获得模型的变更信息。

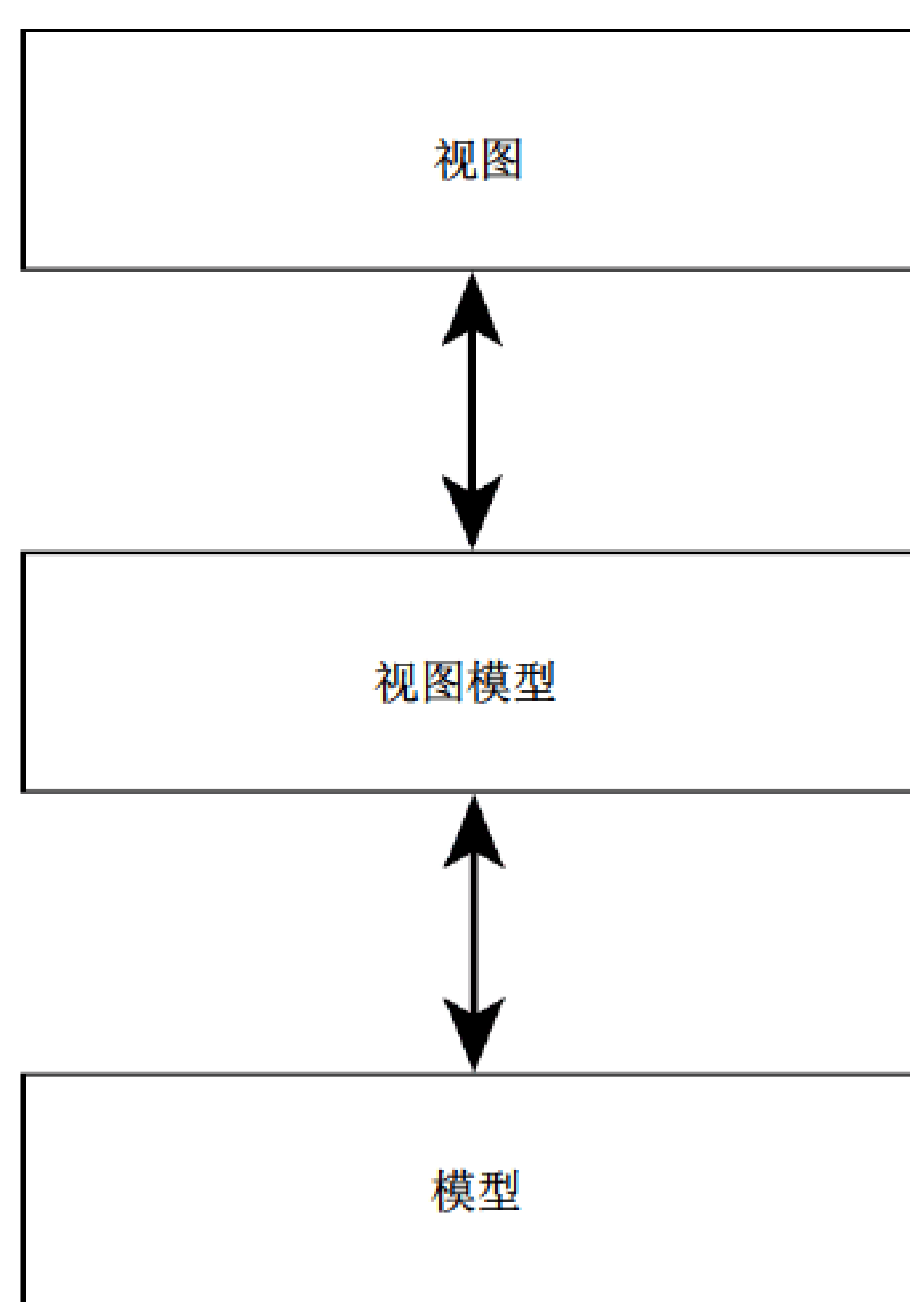



图 34-3

本章的下面几节介绍如何使用这个架构与应用程序创建视图、视图模型、模型和其他需要的模式。

## 34.3　共享代码

在创建这个示例解决方案，开始创建模型之前，需要回过头来看看不同的选项如何在不同的平台之间共享代码。本节讨论不同的选项，考虑需要支持的不同平台和所需要的 API。

### 34.3.1　使用 API 协定和通用 Windows 平台

通用 Windows 平台定义了一个可用于所有 Windows 10 设备的 API。然而，这个 API 在新版本中会改变。使用 Project Properties 中的 Application 设置(参见图 34-4)，可以定义应用程序的目标版本(这是要构建的版本)和系统所需的最低版本。所选 Software Developer Kits (SDK)的版本需要安装在系统上，才能验证哪些 API 可用。为了使用目标版本中最低版本不可用的特性，需要在使用 API 之前，以编程方式检查设备是否支持所需要的具体功能。

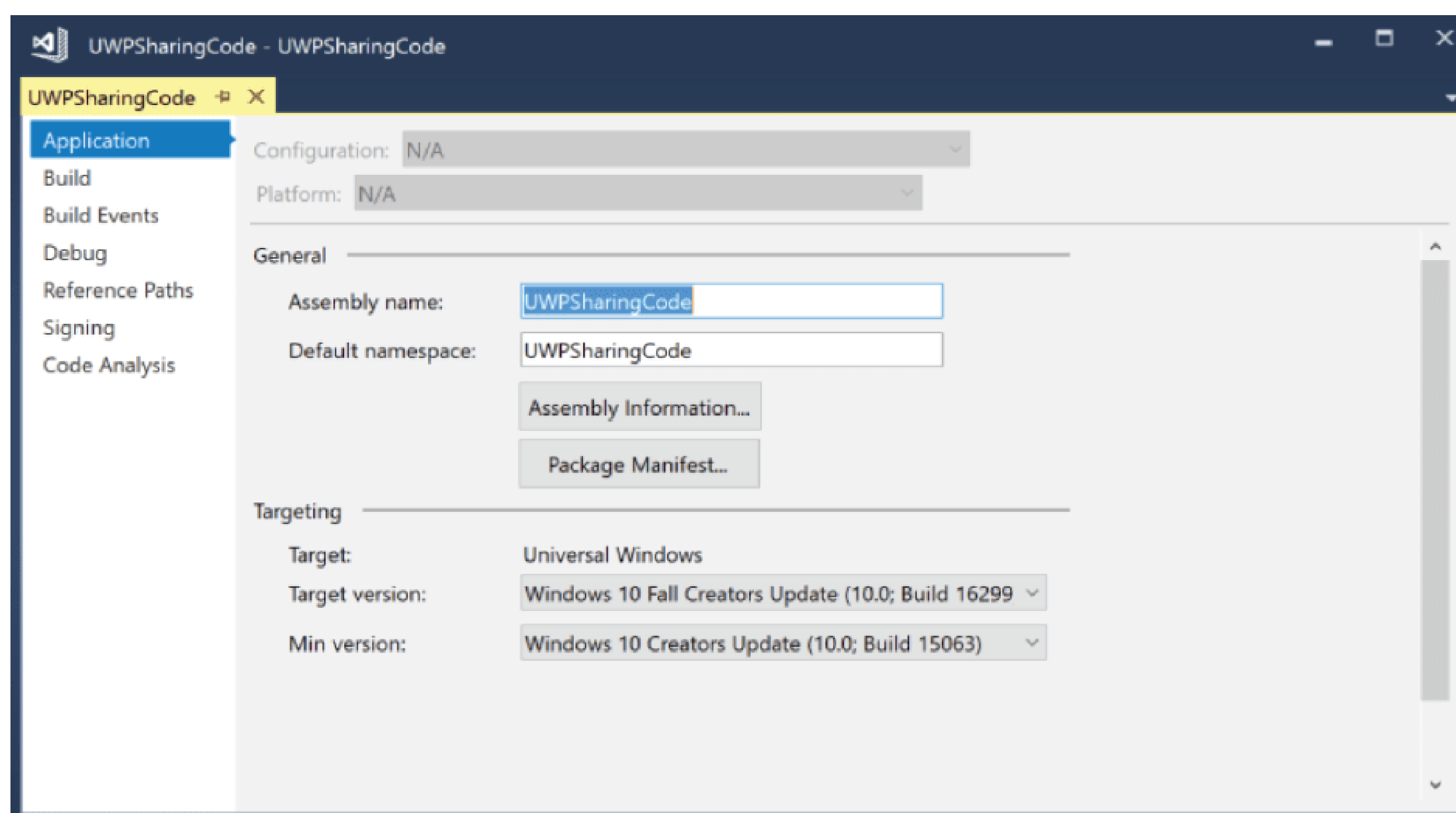



图 34-4

通过 UWP 可以支持不同的设备系列。UWP 定义了几种设备系列：通用、桌面(PC)、手机、物联网(Raspberry Pi)、Surface Hub、Holographic (HoloLens)以及 Xbox。随着时间的推移，会出现更多的设备系列。这些设备系列提供的 API 只能用于这个系列。通过 API 协定指定设备系列的 API。每个设备系列可以提供多个 API 协定。

可以使用设备系列特有的特性，也可以创建运行在所有设备上的二进制图像。通常情况下，应用程序不支持所有的设备系列，可能支持其中的一些设备。为了支持特定的设备系列，使用这些系列的 API，可以在 Solution Explorer 中添加一个 Extension SDK；选择 References | Add Reference，然后选择 Universal Windows | Extensions (参见图 34-5)。在那里可以看到安装的 SDK，并选择需要的 SDK。

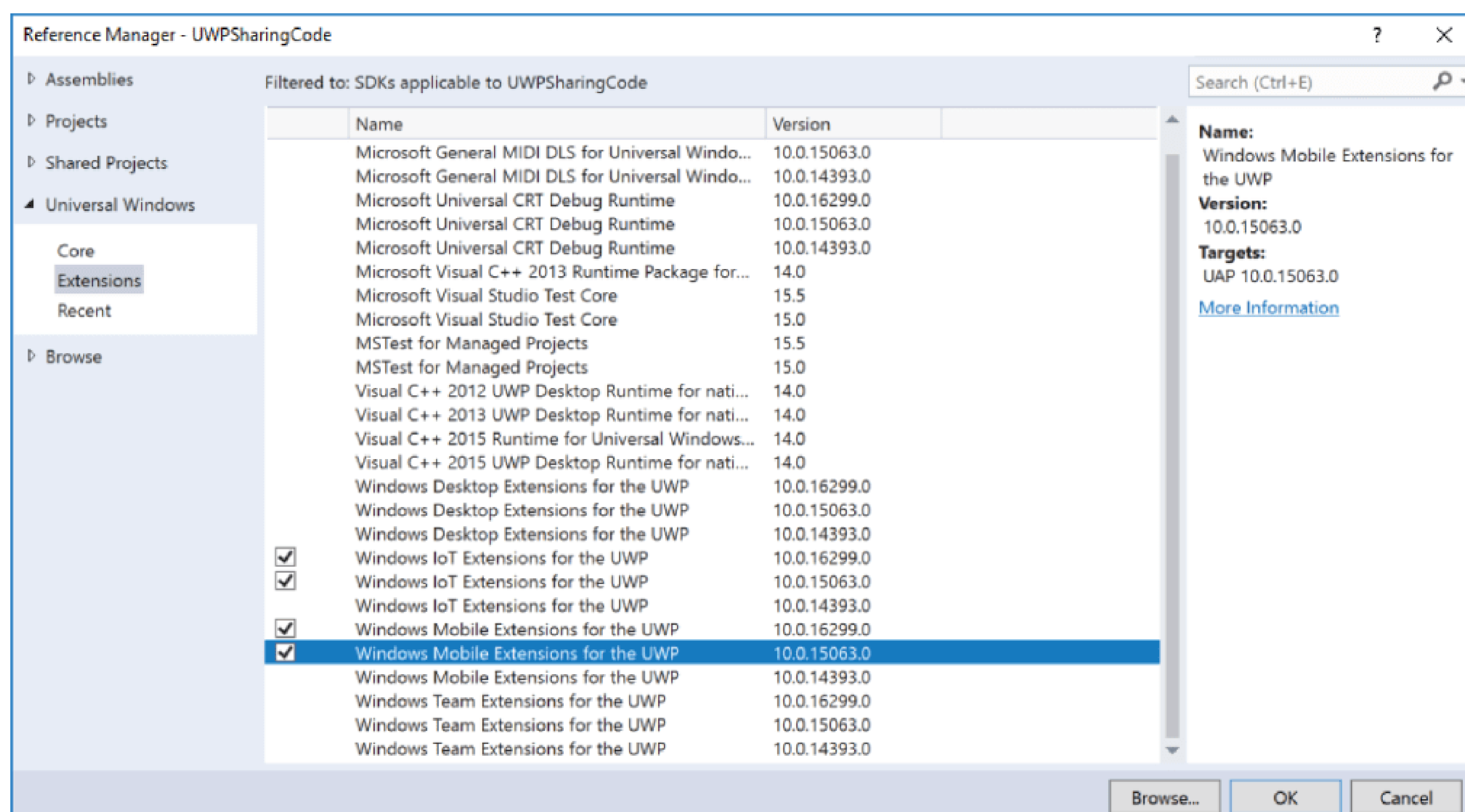



图 34-5

选择 Extension SDK 后，验证 API 协定是否可用，就可以在代码中使用 API。ApiInformation 类(名称空间 Windows.Foundation.Metadata)定义了 IsApiContractPresent 方法，在其中可以检查特定主次版本的 API 协定是否可用。下面的代码片段需要 Windows.Phone.PhoneContract 的主版本 1。如果本协定可用，就可以使用 VibrationDevice：

```
if (ApiInformation.IsApiContractPresent("Windows.Phone.PhoneContract", 1))
{
  VibrationDevice vibration = VibrationDevice.GetDefault();
  vibration.Vibrate(TimeSpan.FromSeconds(1));
}
```

**注意：**
还可以使用 XAML 代码检查可用的 API 协定。条件 XAML 参见第 33 章。

在所有地方检查 API 协定的代码是否非常复杂？其实，如果只针对单一设备系列，就不需要检查 API 是否存在。在前面的示例中，如果应用程序只针对手机，就不需要检查 API。如果针对多个设备平台，就只需要检查特定于设备的 API 调用。可以使用通用的 API 编写有用的应用程序，用于多个设备系列。如果用很多特定于设备的 API 调用支持多个设备系列，建议避免使用 ApiInformation，而应使用依赖注入。

**注意：**
依赖注入(DI)和使用 DI 容器参见本章 34.7.4 节“服务、ViewModel 和依赖注入”。

### 34.3.2　使用共享项目

对 API 协定使用相同的二进制只适用于通用 Windows 平台。如果需要分享代码，就不能使用这个选项，例如，在带有 WPF 的 Windows 桌面应用程序和 UWP 应用程序之间分享代码，或 Xamarin.Forms 应用程序和 UWP 应用程序之间分享代码。在不能使用相同二进制文件的地方创建这些项目类型，就可以使用 Visual Studio 2017 的 Shared Project 模板。

使用 Shared Project 模板与 Visual Studio 创建的项目，没有创建二进制——没有创建程序集。相反，代码在所有引用这个共享项目的项目之间共享。在每个引用共享项目的项目中编译代码。

创建一个类，如下面的代码片段所示，这个类可用于引用共享项目的所有项目。甚至可以通过预处理器指令使用特定于平台的代码。Visual Studio 2017 的 Universal Windows App 模板设置条件编译符号 WINDOWS_UWP，以便将这个符号用于应该只为通用 Windows 平台编译的代码。对于 WPF，通过 WPF 项目把 WPF 添加到条件编译符号中。对于 Xamarin,，可以给条件编译符号添加 XAMARIN。

```
public partial class Demo
{
  public int Id { get; set; }
  public string Title { get; set; }
#if WPF
  public string WPFOnly { get; set; }
#endif
#if WINDOWS_UWP
  public string WindowsAppOnly {get; set; }
#endif
#if XAMARIN
  public string XamarinAppOnly {get; set; }
#endif
}
```

通过 Visual Studio 编辑器编辑共享代码，可以在左上方的栏中选择项目名称，灰显不用于实际项目的部分代码(参见图 34-6)。编辑文件时，智能感知功能还为所选的相应项目提供了 API。

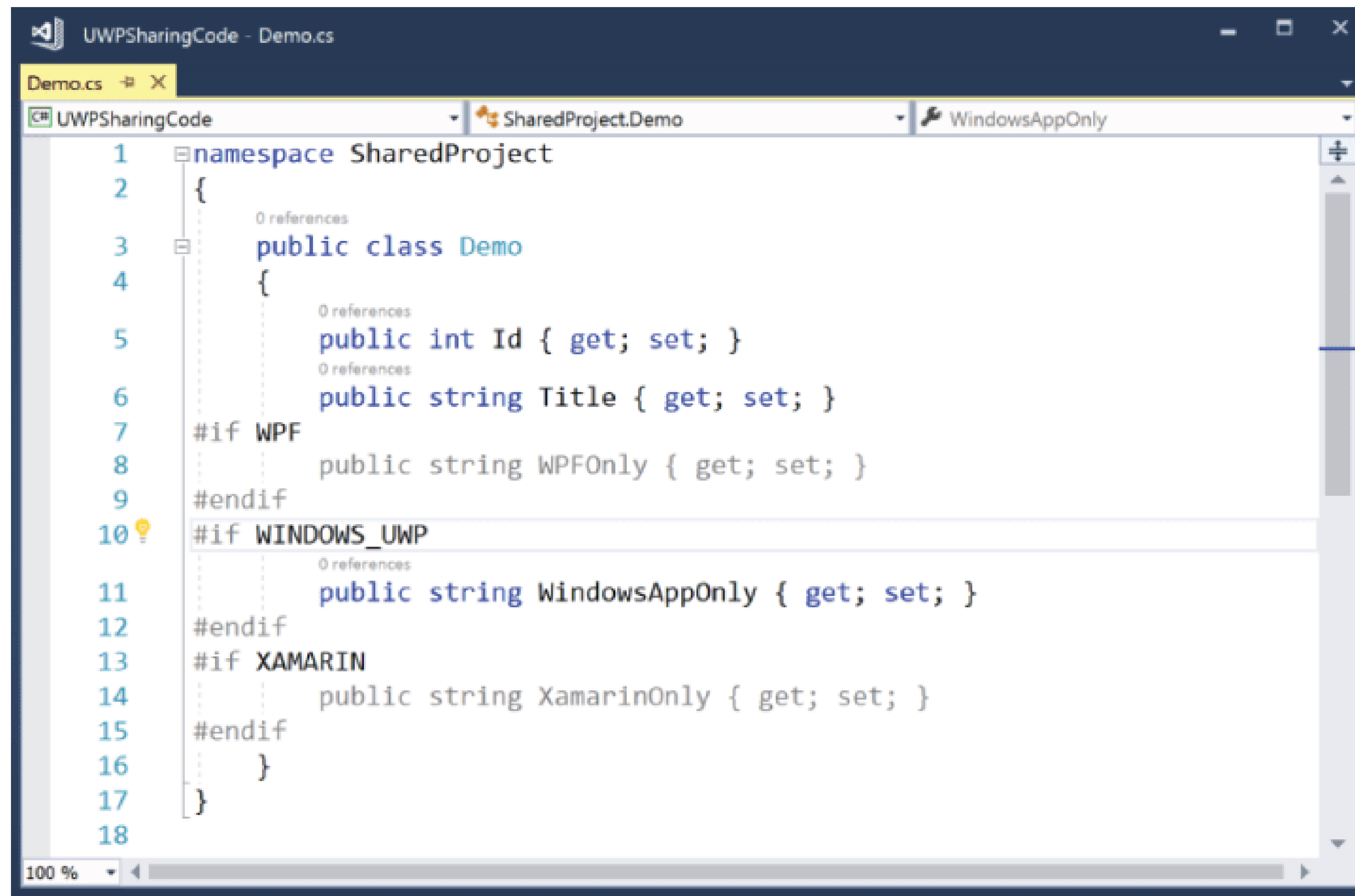



图 34-6

除了使用预处理器指令之外，还可以在 WPF、UWP 或 Xamarin 项目中保留类的不同部分。所以要把类声明为 partial。

**注意：**
C#的 partial 关键字参见第 3 章。

在 WPF 项目中定义相同的类名和相同的名称空间时，就可以扩展共享类。还可以使用基类(假设共享项目没有定义基类)：

```
public class MyBase
{
  //...
}

public partial class Demo: MyBase
{
  public string WPFTitle => $"WPF{Title}";
}
```

### 34.3.3 使用.NET 标准库

共享代码的另一个选择是.NET 标准库。如果所有技术都可以使用.NET 标准，这就是一个简单的任务：创建一个.NET 标准库，就可以在不同的平台之间共享它。只需要关注要使用的.NET 标准的版本就可以了。示例应用程序使用.NET 标准 2.0 库；从构建版本 16299、Xamarin.iOS 10.14、Xamarin.Android 8.0、Xamarin.Mac 3.8 和.NET Framework 4.6.1 开始，它就可以在 Windows 应用程序中使用。

**注意：**
标准库是可移植类库(Portable Class Library，PCL)的替代品，使用起来要容易得多。创建.NET 标准库详见第 19 章。

使用.NET 标准库，每个新版本都有额外的 API。API 永远不会被删除。创建.NET 标准库之后，可以使用 Project Properties 选择应该支持的标准版本(参见图 34-7)。选择它们后，就限制了可用于该版本的 API。

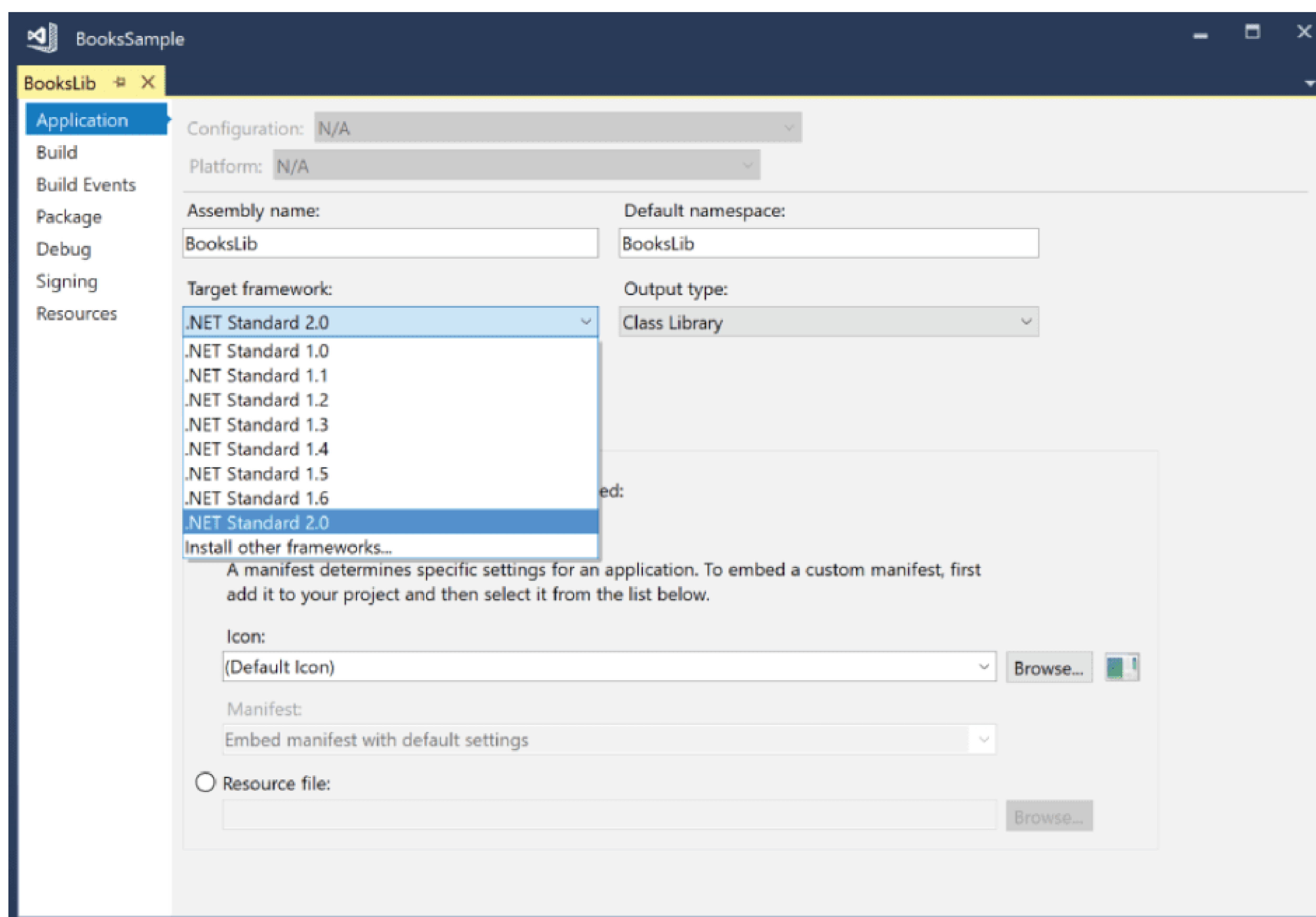



图 34-7

使用共享项目时，可以编写只用于一个平台的代码。要将代码与其他平台区分开来，只能在为这个平台编译代码时，使用预处理器语句来编译代码。.NET 标准库不能这么做。解决这个问题的一个办法是，可以使用标准库为代码定义协定，在需要的地方使用特定于平台的库实现协定。要在特定于平台的库中使用代码和不特定于平台的库，可以使用依赖注入。如何做到这一点是本章更大示例的一部分，参见 34.7 节“视图模型”。

**注意：**

使用.NET 标准库，可以添加对非标准.NET 库的引用。只要不使用在所有目标平台上都不可用的 API 即可。只要使用特定于一个目标平台的 API，其他平台就会崩溃。可以在使用特定的 API 之前添加运行时检查，以避免这个问题。使用特定于平台的代码的更清晰的解决方案是依赖注入，如示例应用程序所示。

## 34.4　示例解决方案

示例解决方案包括一个 Universal Windows Platform 应用程序，用于显示和编辑一个图书列表。在第 37 章中，这款应用将扩展到 iPhone 和 Android 上。为此，解决方案使用如下项目：

- BooksApp——UWP 应用程序项目，是现代应用程序的 UI，此应用程序包含带有 XAML 代码的应用程序视图，以及服务特定于平台的实现。
- BooksLib——一个.NET 标准库 2.0，提供模型、视图模型和服务来创建、读取和更新图书；所有平台都支持.NET 标准 2.0。
- Framework——一个.NET 标准库 2.0，包含可用于所有基于 XAML 的应用程序的类。其中包括视图模型的基类，实现 INotifyPropertyChanged、ICommand 以及其他有用特性的实现代码。

应用程序的用户界面有两个视图：一个视图显示图书列表，一个视图显示图书的详细信息。从列表中选择一本书，就会显示细节，也可以添加和编辑图书。

BooksLib 和 Framework 库可以由带有 XAML 代码的多个应用程序使用——例如，UWP、WPF 和 Xamarin。

Framework 库以可由不同的应用程序使用的方式构建而不仅仅是处理图书的应用程序。BooksLib 中实现了特定于图书的功能。示例应用程序演示了不仅在 UWP 和 Xamarin 之间共享视图模型，还在需要主/从功能的不同应用程序之间共享视图模型。视图模型的基类在 Framework 库中实现。

# 34.5 模型

下面先定义模型，尤其是 Book 类型。这个类型在 UI 中显示和编辑。为了支持数据绑定，在用户界面中更新的属性值需要实现变更通知。BookId 属性只是显示，而不改变，所以变更通知不需要使用这个属性。SetProperty 方法由基类 BindableBase 定义(代码文件 BookslibModels/Book.cs)：

```
public class Book: BindableBase
{
  public int BookId { get; set; }

  private string _title;
  public string Title
  {
    get => _title;
    set => Set(ref _title, value);
  }

  private string _publisher;
  public string Publisher
  {
    get => _publisher;
    set => Set(ref _publisher, value);
  }

  public override string ToString() => Title;
}
```

## 34.5.1 实现变更通知

XAML 元素的对象源需要依赖属性或 INotifyPropertyChanged，才允许更改通知与数据绑定。有了模型类型，才能实现 INotifyPropertyChanged。为了让一个实现可用于不同的项目，实现代码在类 BindableBase 的 Framework 库项目内完成。INotifyPropertyChanged 接口定义了 PropertyChanged 事件。为了触发更改通知，SetProperty 方法实现为一个泛型函数，以支持任何属性类型。在触发通知之前，检查新值是否与当前值不同(代码文件 Framework/BindableBase.cs)：

```
public abstract class BindableBase: INotifyPropertyChanged
{
  public event PropertyChangedEventHandler PropertyChanged;

  protected virtual void OnPropertyChanged(
    [CallerMemberName] string propertyName = null) =>
      PropertyChanged?.Invoke(this,
        new PropertyChangedEventArgs(propertyName));

  protected virtual bool SetProperty<T>(ref T item, T value,
    [CallerMemberName] string propertyName = null)
  {
    if (EqualityComparer<T>.Default.Equals(item, value)) return false;
    item = value;
    OnPropertyChanged(propertyName);
    return true;
  }
}
```

**注意：**
依赖属性参见第 33 章。

### 34.5.2　使用 Repository 模式

接下来，需要一种方法来检索、更新和删除 Book 对象。使用 EF Core 可以在数据库中读写图书。虽然 EF Core 可以在通用 Windows 平台上访问，但通常这是一个后台任务，因此本章未涉及。为了使后端可以在客户端应用程序中访问，在服务器端选择 Web API 技术。这些主题参见第 26 章和第 32 章。在客户端应用程序中，最好能独立于数据存储。为此，定义 Repository 设计模式。Repository 模式是模型和数据访问层之间的中介，它可以作为对象的内存集合。它抽象出了数据访问层，使单元测试更方便。

通用接口 IQueryRepository 定义的方法通过 ID 获取一项，或获取一个条目列表(代码文件 Bookslib/Services/IQueryRepository.cs)：

```
public interface IQueryRepository<T, in TKey>
  where T: class
{
  Task<T> GetItemAsync(TKey id);
  Task<IEnumerable<T>> GetItemsAsync();
}
```

通用接口IUpdateRepository定义方法来添加、更新和删除条目(代码文件Bookslib/Services/IUpdateRepository.cs)：

```
public interface IUpdateRepository<T, in TKey>
  where T: class
{
  Task<T> AddAsync(T item);
  Task<T> UpdateAsync(T item);
  Task<bool> DeleteAsync(TKey id);
}
```

IBooksRepository 接口为泛型类型 T 定义 Book 类型，使前两个泛型接口更具体(代码文件 Bookslib/Services/IBooksRepository.cs):

```
public interface IBooksRepository: IQueryRepository<Book, int>,
  IUpdateRepository<Book, int>
{
}
```

使用这些接口，可以改变存储库。创建一个示例库 BooksSampleRepository，它实现接口 IBooksRepository 的成员，包含一个图书的初始列表(代码文件 Bookslib/Services/BooksSampleRepository.cs)：

```
public class BooksSampleRepository: IBooksRepository
{
  private List<Book> _books;
  public BooksRepository()
  {
    InitSampleBooks();
  }

  private void InitSampleBooks()
  {
    _books = new List<Book>()
    {
      new Book
      {
        BookId = 1,
        Title = "Professional C# 7 and .NET Core 2",
        Publisher = "Wrox Press"
      },
      new Book
      {
        BookId = 2,
        Title = "Professional C# 6 and .NET Core 1.0",
        Publisher = "Wrox Press"
      },
      new Book
      {
        BookId = 3,
        Title = "Professional C# 5.0 and .NET 4.5.1",
        Publisher = "Wrox Press"
      },
      new Book
      {
```

```
            BookId = 4,
            Title = "Enterprise Services with the .NET Framework",
            Publisher = "AWL"
          }
        };
      }

      public Task<bool> DeleteAsync(int id)
      {
        Book bookToDelete = _books.Find(b => b.BookId == id);
        if (bookToDelete != null)
        {
          return Task.FromResult<bool>(_books.Remove(bookToDelete));
        }
        return Task.FromResult<bool>(false);
      }

      public Task<Book> GetItemAsync(int id) =>
        Task.FromResult(_books.Find(b => b.BookId == id));

      public Task<IEnumerable<Book>> GetItemsAsync() =>
        Task.FromResult<IEnumerable<Book>>(_books);

      public Task<Book> UpdateAsync(Book item)
      {
        Book bookToUpdate = _books.Find(b => b.BookId == item.BookId);
        int ix = _books.IndexOf(bookToUpdate);
        _books[ix] = item;
        return Task.FromResult(_books[ix]);
      }

      public Task<Book> AddAsync(Book item)
      {
        item.BookId = _books.Select(b => b.BookId).Max() + 1;
        _books.Add(item);
        return Task.FromResult(item);
      }
    }
```

**注意：**

*存储库定义了异步方法，但这里不需要它们，因为书的检索和更新只在内存中进行。方法定义为异步，是因为用于访问 ASP.NET Core Web API 的存储库在本质上是异步的。*

# 34.6 服务

要从存储库中获取图书，需要使用一个服务，并且可以在访问相同数据的多个视图模型中使用它。因此，服务是在视图模型之间共享数据的好地方。

图书的示例服务实现了泛型接口 IItemsService。这个接口定义了类型 ObservableCollection 的 Items 属性。当集合发生变化时，ObservableCollection 实现了用于通知的 INotifyCollectionChanged 接口。接口 IItemsService 也定义了 SelectedItem 属性，并使用事件 SelectedItemChanged 更改通知。除此之外，RefreshAsync、AddOrUpdateAsync 和 DeleteAsync 都是需要由服务类实现的方法(代码文件 Framework/Services/IItemsService.cs):

```
public interface IItemsService<T>
{
  Task RefreshAsync();

  Task<T> AddOrUpdateAsync(T item);

  Task DeleteAsync(T item);

  ObservableCollection<T> Items { get; }

  T SelectedItem { get; set; }
  event EventHandler<T> SelectedItemChanged;
}
```

类 BooksService 派生自基类 BindableBase，并实现了泛型接口 IItemsService。BooksService 使用以前创建的

SampleBooksRepository，但只需要 IBooksRepository 接口提供的这个类的功能。该类通过构造函数注入，用于刷新图书列表、添加或更新图书以及删除图书(代码文件 BooksLib/Services/BooksService.cs)：

```
public class BooksService : BindableBase, IItemsService<Book>
{
  private ObservableCollection<Book> _books = new ObservableCollection<Book>();
  private readonly IBooksRepository _booksRepository;

  public event EventHandler<Book> SelectedItemChanged;

  public BooksService(IBooksRepository repository)
  {
    _booksRepository = repository;
  }

  public ObservableCollection<Book> Items => _books;

  private Book _selectedItem;
  public Book SelectedItem
  {
    get => _selectedItem;
    set
    {
      if (Set(ref _selectedItem, value))
      {
        SelectedItemChanged?.Invoke(this, _selectedItem);
      }
    }
  }

  public async Task<Book> AddOrUpdateAsync(Book book)
  {
    Book updated = null;
    if (book.BookId == 0)
    {
      updated = await _booksRepository.AddAsync(book);
    }
    else
    {
      updated = await _booksRepository.UpdateAsync(book);
    }
    return updated;
  }

  public Task DeleteAsync(Book book) =>
    _booksRepository.DeleteAsync(book.BookId);

  public async Task RefreshAsync()
  {
    IEnumerable<Book> books = await _booksRepository.GetItemsAsync();
    _books.Clear();
    foreach (var book in books)
    {
      _books.Add(book);
    }
    SelectedItem = Items.FirstOrDefault();
  }
}
```

既然服务功能已经就绪，下面就继续讨论视图模型。

## 34.7　视图模型

每个视图或页面都有一个视图模型。在示例应用程序中，BooksPage 与 BooksViewModel 关联。在后面的示例中，用户控件也可以有其特定的视图模型，但这并不总是必需的。BookDetailPage 与 BookDetailViewModel 关联。如果书的列表和细节可以在同一个页面中实现，这就是一个 UI 设计决策。这取决于应用程序的可用屏幕大小。屏幕上可以放什么？对于示例应用程序，采用了一种灵活的方法。如果应用程序可用的空间足够大，则 BooksPage 显示列表和详细信息；如果空间不够大，则数据将显示在多个单独的页面中，其中包含导航。

页面视图和视图模型之间是一对一映射。实际上，视图和视图模型之间还有多对一映射，因为视图存在于

不同的技术中——WPF、UWP 和 Xamarin。视图模型必须对视图一无所知，但视图要了解视图模型。视图模型用.NET 标准库实现，这样就可以把它用于许多技术。

对于视图模型的通用功能，创建基类是有意义的。Framework 库包含一个 ViewModelBase 类，该类为进度信息和错误实现了特性(代码文件 Framework/ViewModels/ViewModelBase.cs)：

```
public abstract class ViewModelBase : BindableBase
{
  // functionality for progress information and
  // error information
}
```

示例应用程序显示了图书列表，并允许用户选择图书。在这里，为具有属性 Items 和 SelectedItem 的视图模型定义泛型基类是有用的。这些属性的实现利用了先前创建的服务来实现 IItemsService 接口 (代码文件 Framework/ViewModels/MasterDetailViewModel.cs)：

```
public abstract class MasterDetailViewModel<TItemViewModel, TItem> :
  ViewModelBase
  where TItemViewModel : IItemViewModel<TItem>
  where TItem: class
{
  private readonly IItemsService<TItem> _itemsService;

  public MasterDetailViewModel(IItemsService<TItem> itemsService)
  {
    _itemsService = itemsService;

    //...
  }


  public ObservableCollection<TItem> Items => _itemsService.Items;

  protected TItem _selectedItem;
  public virtual TItem SelectedItem
  {
    get => _itemsService.SelectedItem;
    set
    {
      if (!EqualityComparer<TItem>.Default.Equals(
        _itemsService.SelectedItem, value))
      {
        _itemsService.SelectedItem = value;
        OnPropertyChanged();
      }
    }
  }

  //...

}
```

**注意：**
泛型参见第 5 章。

要详细显示一项，基类 ItemViewModel 定义了 Item 属性(代码文件 Framework/ViewModels/ItemViewModel.cs)：

```
public abstract class ItemViewModel<T> : ViewModelBase, IItemViewModel<T>
{
    private T _item;
    public virtual T Item
    {
        get => _item;
        set => Set(ref _item, value);
    }
}
```

比简单类 ItemViewModel 更复杂的是视图模型类 EditableItemViewModel。这个类通过允许编辑来扩展 ItemViewModel，因此它定义了读取或编辑模式。属性 IsReadMode 只是 IsEditMode 的逆属性。

EditableItemViewModel 使用与 MasterDetailViewModel 类相同的服务，该服务实现了接口 IItemsService。这样，EditableItemViewModel 和 MasterDetailViewModel 类就可以共享相同的项和相同的选择。视图模型类允许用户取消输入。为此，项通过 EditItem 属性具有复制版本(代码文件 Framework/ViewModels/EditableItemViewModel.cs)：

```
public abstract class EditableItemViewModel<TItem> : ItemViewModel<TItem>,
  IEditableObject
  where TItem : class
{
  private readonly IItemsService<TItem> _itemsService;

  public EditableItemViewModel(IItemsService<TItem> itemsService)
  {
    _itemsService = itemsService;
    Item = _itemsService.SelectedItem;

    PropertyChanged += (sender, e) =>
    {
      if (e.PropertyName == nameof(Item))
      {
        OnPropertyChanged(nameof(EditItem));
      }
    };
    //...
  }

  //...
  private bool _isEditMode;
  public bool IsReadMode => !IsEditMode;
  public bool IsEditMode
  {
    get => _isEditMode;
    set
    {
      if (Set(ref _isEditMode, value))
      {
        OnPropertyChanged(nameof(IsReadMode));
        //...
      }
    }
  }

  private TItem _editItem;
  public TItem EditItem
  {
    get => _editItem ?? Item;
    set => Set(ref _editItem, value);
  }
  //...
}
```

### 34.7.1　使用 IEditableObject

接口 IEditableObject 定义方法来在不同编辑状态之间更改对象。这个接口在名称空间 System.ComponentModel 中定义。IEditableObject 定义了方法 BeginEdit、CancelEdit 和 EndEdit。调用 BeginEdit 来将项从读取模式更改为编辑模式。CancelEdit 取消编辑并切换回“读取”模式。EndEdit 是用于编辑模式的成功结束，因此需要保存数据。EditableItemViewModel 类通过切换编辑模式、创建项的副本并保存状态来实现这个接口的方法。这个视图模型类是一个泛型类，不知道如何复制和保存该项。通过使用二进制串行化可以进行复制。然而，并不是所有的对象都支持二进制序列化。相反，实现代码转发到派生自 EditableItemViewModel 的类中，类似于保存方法 OnSaveAsync。OnSaveAsync 和 CreateCopy 定义为抽象方法，因此需要由派生类实现。另一个方法 OnEndEditAsync 定义为在 CancelEdit 和 EndEdit 结尾调用。这个方法可以由派生类实现，但是没有必要这样做。这就是为什么方法声明为空的原因(代码文件 Framework/ViewModels/EditableItemViewModel.cs)：

```
public virtual void BeginEdit()
{
  IsEditMode = true;
  TItem itemCopy = CreateCopy(Item);
  if (itemCopy != null)
  {
```

```
      EditItem = itemCopy;
    }
  }

  public async virtual void CancelEdit()
  {
    IsEditMode = false;
    EditItem = default(TItem);
    await _itemsService.RefreshAsync();
    await OnEndEditAsync();
  }

  public async virtual void EndEdit()
  {
    using (StartInProgress())
    {
      await OnSaveAsync();
      EditItem = default(TItem);
      IsEditMode = false;
      await _itemsService.RefreshAsync();
      await OnEndEditAsync();
    }
  }

  public abstract Task OnSaveAsync();
  public abstract TItem CreateCopy(TItem item);
  public virtual Task OnEndEditAsync() => Task.CompletedTask;
```

### 34.7.2 视图模型的具体实现

下面继续讨论视图模型的具体实现。BookDetailViewModel 派生自 EditableItemViewModel，并将 Book 指定为泛型参数。由于基类已经实现了主要功能，因此这个类可以很简单。它为接口 IItemsService 和 INavigationSerivce 注入服务。在 OnSaveAsync 方法中，请求转发到 IItemsService。OnSaveAsync 方法还使用了 ILogger 和 IMessageService 接口。在视图模型类中，CreateCopy 方法实现图书副本的创建。这个方法由基类调用(代码文件 BooksLib/ViewModels/BookDetailViewModel.cs)：

```
public class BookDetailViewModel : EditableItemViewModel<Book>
{
  private readonly IItemsService<Book> _itemsService;
  private readonly INavigationService _navigationService;
  private readonly IMessageService _messageService;
  private readonly ILogger _logger;

  public BookDetailViewModel(IItemsService<Book> itemsService,
    INavigationService navigationService, IMessageService messageService,
    ILogger<BookDetailViewModel> logger)
    : base(itemsService)
  {
    _itemsService = itemsService;
    _navigationService = navigationService;
    _messageService = messageService;
    _logger = logger;

    itemsService.SelectedItemChanged += (sender, book) =>
    {
      Item = book;
    };
  }

  public bool UseNavigation { get; set; }

  public override Book CreateCopy(Book item) =>
    new Book
    {
      BookId = item?.BookId ?? -1,
      Title = item?.Title ?? "enter a title",
      Publisher = item?.Publisher ?? "enter a publisher"
    };

  public override async Task OnSaveAsync()
  {
    try
    {
```

```
        await _itemsService.AddOrUpdateAsync(EditItem);
      }
      catch (Exception ex)
      {
        _logger.LogError("error {0} in {1}", ex.Message, nameof(OnSaveAsync));
        await _messageService.ShowMessageAsync("Error saving the data");
      }
    }
    //...
  }
```

**注意：**

ILogger 接口参见第 29 章。接口 IMessageService 将在本章后面的 34.8.1 节“从视图模型中打开对话框”中讨论。导航服务的定义和使用将在本章后面的 34.8.2 节“页面之间的导航”中讨论。

类 BooksViewModel 可以通过继承 MasterDetailViewModel 的主要功能来保持简单。这个类只是注入稍后讨论的 INavigationService 接口，将 IItemsService 接口转发给基类，并重写由基类调用的 OnAdd 方法(代码文件 BooksLib/ViewModels/BooksViewModel.cs)：

```
public class BooksViewModel : MasterDetailViewModel<BookItemViewModel, Book>
{
  private readonly IItemsService<Book> _booksService;
  private readonly INavigationService _navigationService;

  public BooksViewModel(IItemsService<Book> booksService,
    INavigationService navigationService)
    : base(booksService)
  {
    _booksService = booksService ??
      throw new ArgumentNullException(nameof(booksService));
    _navigationService = navigationService ??
      throw new ArgumentNullException(nameof(navigationService));
    //...
  }

  public override void OnAdd()
  {
    var newBook = new Book();
    Items.Add(newBook);
    SelectedItem = newBook;
  }

  //...
}
```

### 34.7.3　命令

视图模型提供了实现 ICommand 接口的命令。命令允许通过数据绑定来分离视图和命令处理程序方法。命令还提供启用或禁用命令的功能。ICommand 接口定义了方法 Execute 和 CanExecute，以及 CanExecuteChanged 事件。

要将命令映射到方法，在 Framework 库中定义了 RelayCommand 类。

RelayCommand 定义了两个构造函数，其中一个委托可以传递应通过命令调用的方法，另一个委托定义了命令是否可用(代码文件 Framework/RelayCommand.cs)：

```
public class RelayCommand: ICommand
{
  private readyonly Action _execute;
  private readonly Func<bool> _canExecute;

  public RelayCommand(Action execute, Func<bool> canExecute)
  {
    _execute = execute ?? throw new ArgumentNullException(nameof(execute));
    _canExecute = canExecute;
  }

  public RelayCommand(Action execute)
    : this(execute, null)
```

```
  { }

  public event EventHandler CanExecuteChanged;

  public bool CanExecute(object parameter) => _canExecute?.Invoke() ?? true;

  public void Execute(object parameter) => _execute();
}
```

EditableItemViewModel 的构造函数创建新的 RelayCommand 对象，在执行命令时，指定前面所示的方法 BeginEdit、CancelEdit 和 EndEdit。所有这些命令还使用 IsReadMode 和 IsEditMode 属性来检查该命令是否可用。当 IsEditMode 属性发生更改时，将触发命令的 CanExecuteChanged 事件，相应地更新命令(代码文件 Framework/ViewModels/EditableItemViewModel.cs)：

```
public abstract class EditableItemViewModel<TItem> : ItemViewModel<TItem>,
  IEditableObject
  where TItem : class
{
  private readonly IItemsService<TItem> _itemsService;

  public EditableItemViewModel(IItemsService<TItem> itemsService)
  {
    _itemsService = itemsService;
    Item = _itemsService.SelectedItem;

    EditCommand = new RelayCommand(BeginEdit, () => IsReadMode);
    CancelCommand = new RelayCommand(CancelEdit, () => IsEditMode);
    SaveCommand = new RelayCommand(EndEdit, () => IsEditMode);
  }

  public RelayCommand EditCommand { get; }
  public RelayCommand CancelCommand { get; }
  public RelayCommand SaveCommand { get; }

  //...

  public bool IsEditMode
  {
    get => _isEditMode;
    set
    {
      if (Set(ref _isEditMode, value))
      {
        OnPropertyChanged(nameof(IsReadMode));
        CancelCommand.OnCanExecuteChanged();
        SaveCommand.OnCanExecuteChanged();
        EditCommand.OnCanExecuteChanged();
      }
    }
  }
  //...
}
```

从 XAML 代码中，命令绑定到按钮的 Command 属性。在创建视图时，将对此进行更详细的讨论(代码文件 BooksApp/Views/BookDetailUserControl.xaml)：

```
<AppBarButton Content="Edit" Icon="Edit"
  Command="{x:Bind ViewModel.EditCommand, Mode=OneTime}" />
<AppBarButton Content="Save" Icon="Save"
  Command="{x:Bind ViewModel.SaveCommand, Mode=OneTime}" />
<AppBarButton Content="Cancel" Icon="Cancel"
  Command="{x:Bind ViewModel.CancelCommand, Mode=OneTime}" />
```

### 34.7.4 服务、ViewModel 和依赖注入

视图模型和服务注入服务，需要创建视图模型。为此，可以使用依赖注入容器。样例应用程序使用 Microsoft.Extensions.DependencyInjection。该容器在 ApplicationServices 类中配置。这个类是用单例模式实现的。在构造函数中，配置 DI 容器。属性 ServiceProvider 返回可检索服务的容器(代码文件 BooksApp/App.xaml.cs)：

```
public class ApplicationServices
{
  private ApplicationServices()
```

```
  {
    var services = new ServiceCollection();
    services.AddSingleton<IBooksRepository, BooksSampleRepository>();
    services.AddSingleton<IItemsService<Book>, BooksService>();
    services.AddTransient<BooksViewModel>();
    services.AddTransient<BookDetailViewModel>();
    services.AddTransient<MainPageViewModel>();
    services.AddSingleton<IMessageService, UWPMessageService>();
    services.AddSingleton<INavigationService, UWPNavigationService>();
    services.AddSingleton<UWPInitializeNavigationService>();
    services.AddLogging(builder =>
    {
#if DEBUG
      builder.AddDebug();
#endif
    });
    ServiceProvider = services.BuildServiceProvider();
  }

  private static ApplicationServices _instance;
  private static object _instanceLock = new object();
  private static ApplicationServices GetInstance()
  {
    lock (_instanceLock)
    {
      return _instance ?? (_instance = new ApplicationServices());
    }
  }

  public static ApplicationServices Instance => _instance ?? GetInstance();

  public IServiceProvider ServiceProvider { get; }
}
```

现在视图模型需要与视图相关联，为此，在 BooksPage 中访问 App 类的 AppServices 属性，并从 DI 容器中调用 GetService 方法。然后，容器用视图模型类的构造函数中定义的所需服务实例化视图模型类。BooksPage 包含了一个用户控件，以获取需要不同视图模型的图书的详细信息。此视图模型是通过设置 BookDetailUC 用户控件的 ViewModel 属性来分配的(代码文件 BooksApp/Views/BooksPage.xaml.cs)：

```
public sealed partial class BooksPage : Page
{
  public BooksPage()
  {
    this.InitializeComponent();
    ViewModel.UseNavigation = false;
    BookDetailUC.ViewModel =
      ApplicationServices.Instance.ServiceProvider
       .GetService<BookDetailViewModel>();
  }

  public BooksViewModel ViewModel { get; } =
    (Application.Current as App).AppServices.GetService<BooksViewModel>();
```

在 BookDetailPage 中，与视图模型的关联是类似的(代码文件 BooksApp/Views/BookDetailPage.xaml.cs)：

```
public sealed partial class BookDetailPage : Page
{
  public BookDetailPage()
  {
    this.InitializeComponent();
    ViewModel.UseNavigation = true; // if the Page is used, enable navigation
  }

  public BookDetailViewModel ViewModel { get; } =
    ApplicationServices.Instance.ServiceProvider
      .GetService<BookDetailViewModel>();
}
```

## 34.8　视图

前面介绍了视图模型的创建，并将视图连接到视图模型，现在就该介绍视图了。

应用程序的主视图由 MainPage 定义。该页面使用的是 Windows 10 update 16299 (Fall Creators Update) 中新

增)的 NavigationView 控件。通常，如果只有一个导航项的小列表，就不应该使用这个 UI 控件。但是，在示例应用程序中使用了控件，因为假定应用程序中的选项将增长到 8 个以上。

NavigationView 控件将 SelectionChanged 事件分配给 MainPageViewModel 的 OnNavigationSelectionChanged 方法。这个视图模型与其他模型非常不同，将在 34.8.2 节“页面之间的导航”中讨论。定义一个 NavigationViewItem，以导航到 BooksPage(代码文件 BooksApp/MainPage.xaml)：

```
<NavigationView Background=
  "{ThemeResource ApplicationPageBackgroundThemeBrush}"
  SelectionChanged="{x:Bind ViewModel.OnNavigationSelectionChanged,
    Mode=OneTime}">
  <NavigationView.MenuItems>
    <NavigationViewItem Content="Books" Tag="books">
      <NavigationViewItem.Icon>
        <FontIcon FontFamily="Segoe MDL2 Assets" Glyph="&#xE82D;" />
      </NavigationViewItem.Icon>
    </NavigationViewItem>
  </NavigationView.MenuItems>

  <Frame x:Name="ContentFrame" Margin="24">
    <Frame.ContentTransitions>
      <TransitionCollection>
        <NavigationThemeTransition/>
      </TransitionCollection>
    </Frame.ContentTransitions>
  </Frame>
</NavigationView>
```

**注意：**
NavigationView 控件参见第 33 章。

图 34-8 显示了正在运行的应用程序的 NavigationView，其中包含指向 BooksPage 的导航项。

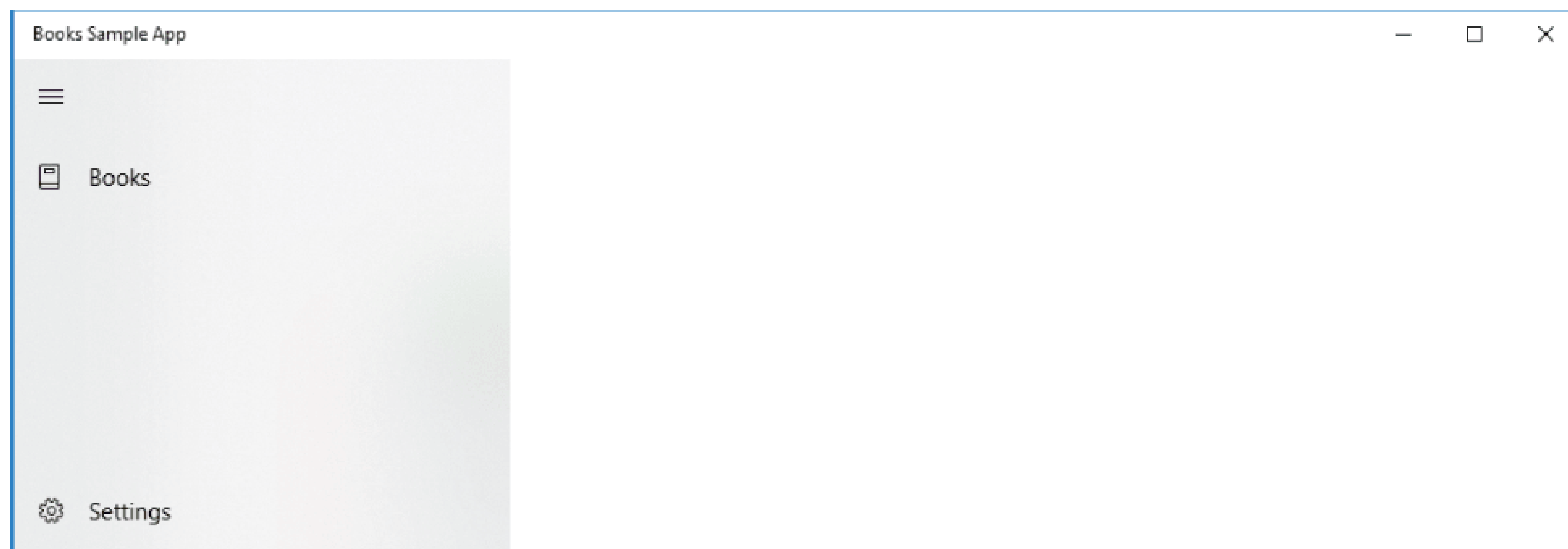



图 34-8

BooksPage 包含一个 ListView，并将 ItemsSource 绑定到 BooksViewModel 的 ItemsViewModel 属性上。通常将它绑定到 BooksViewModel 的 Items 属性上。然而，单个列表项不仅用于显示 Book 对象的值，还包含绑定到命令的按钮。要实现这样的功能，该项将使用另一个视图模型(代码文件 BooksApp/Views/BooksPage.xaml)：

```
<StackPanel Orientation="Horizontal" Grid.Row="1">
  <AppBarButton Icon="Refresh" IsCompact="True"
    Command="{x:Bind ViewModel.RefreshCommand}"
    Label="Get Books" />
  <AppBarButton Icon="Add" IsCompact="True"
    Command="{x:Bind ViewModel.AddCommand}"
    Label="Add Book" />
</StackPanel>
<ListView ItemTemplate="{StaticResource BookItemTemplate}" Grid.Row="2"
  ItemsSource="{x:Bind ViewModel.ItemsViewModels, Mode=OneWay}"
  SelectedItem="{x:Bind ViewModel.SelectedItemViewModel, Mode=TwoWay}" >
</ListView>
<views:BookDetailUserControl x:Name="BookDetailUC" Visibility="Collapsed"
  Grid.Column="1" Grid.Row="1" Grid.RowSpan="2" />
```

BookDetailPage 仅仅包含一个用户控件 BookDetailUserControl。BookDetailPage 关联了 BookDetailViewModel，如前所述。将 BookDetailPage 的 ViewModel 属性分配给 BookDetailUserControl 的 ViewModel 属性，把这个视图模型转发到 BookDetailUserControl (代码文件 BooksApp/Views/BookDetailPage.xaml)：

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
  <views:BookDetailUserControl ViewModel="{x:Bind ViewModel, Mode=OneTime}" />
</Grid>
```

BookDetailUserControl 的依赖属性显示在下面的代码片段中。然后，将对该视图模型的映射用于 XAML 代码中的数据绑定(代码文件 BooksApp/Views/BookDetailUserControl.xaml.cs)：

```
public BookDetailViewModel ViewModel
{
  get => (BookDetailViewModel)GetValue(ViewModelProperty);
  set => SetValue(ViewModelProperty, value);
}

public static readonly DependencyProperty ViewModelProperty =
  DependencyProperty.Register("ViewModel", typeof(BookDetailViewModel),
  typeof(BookDetailUserControl), new PropertyMetadata(null));
```

BookDetailUserControl 的用户界面使用了两个 StackPanel 元素。对于第一个 StackPanel, AppBarButton 控件的 Command 属性绑定到视图模型中定义的 EditCommand、SaveCommand 和 CancelCommand 命令。按钮将根据命令的状态自动启用或禁用。在第二个 StackPanel 中，TextBox 元素用于显示 Book 的 Title 和 Publisher 属性。对于只读显示，把 IsReadOnly 属性分配给视图模型的 IsReadMode 属性。当视图模型设置为编辑模式时，TextBox 控件允许输入数据(代码文件 BooksApp/Views/BookDetailUserControl.xaml)：

```
<StackPanel Orientation="Horizontal">
  <AppBarButton Content="Edit" Icon="Edit"
    Command="{x:Bind ViewModel.EditCommand, Mode=OneTime}" />
  <AppBarButton Content="Save" Icon="Save"
    Command="{x:Bind ViewModel.SaveCommand, Mode=OneTime}" />
  <AppBarButton Content="Cancel" Icon="Cancel"
    Command="{x:Bind ViewModel.CancelCommand, Mode=OneTime}" />
</StackPanel>
<StackPanel Orientation="Vertical" Grid.Row="1">
  <TextBox Header="Title"
        IsReadOnly="{x:Bind ViewModel.IsReadMode, Mode=OneWay}"
        Text="{x:Bind ViewModel.EditItem.Title, Mode=TwoWay,
         UpdateSourceTrigger=PropertyChanged}" />
  <TextBox Header="Publisher"
         IsReadOnly="{x:Bind ViewModel.IsReadMode, Mode=OneWay}"
         Text="{x:Bind ViewModel.EditItem.Publisher, Mode=TwoWay,
          UpdateSourceTrigger=PropertyChanged}" />
</StackPanel>
```

图 34-9 显示了在 ListView 中使用图书的运行应用程序。在 BookDetailUserControl 中，当前禁用 Save 和 Cancel 命令，启用 Edit 命令。

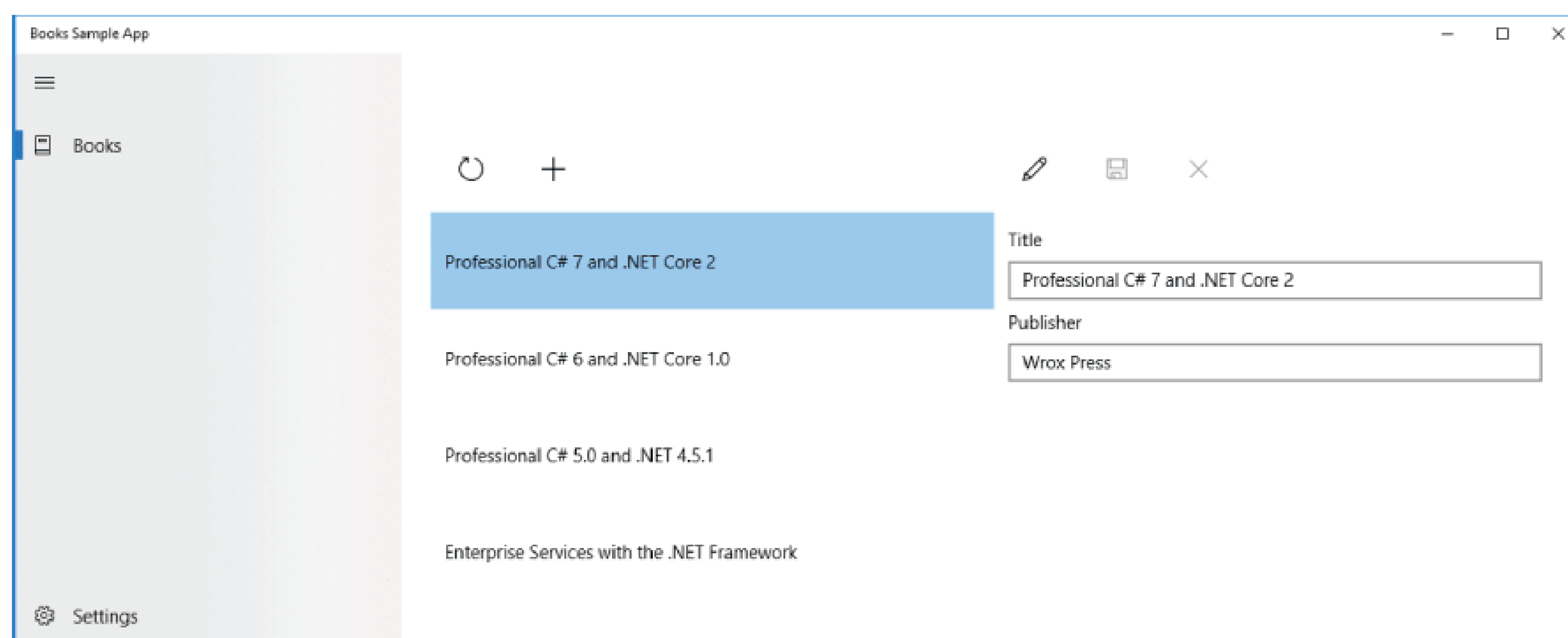



图 34-9

### 34.8.1 从视图模型中打开对话框

有时需要在视图模型中的操作显示对话框。由于视图模型是在.NET 标准库中实现的，因此不可能从 UWP 中访问 MessageDialog 类。无论如何，都应该避免这种情况，因为 MessageDialog 是特定于 UWP 的。在 WPF 中，可以使用 MessageBox 类。在 Xamarin.Forms 中，可以使用 Page.DisplayAlert。

我们需要定义一个可以由视图模型和服务使用的协定。该协定在 BooksLib 库中用 IMessageService 接口定义(代码文件 BooksLib/Services/IMessageService.cs)：

```
public interface IMessageService
{
  Task ShowMessageAsync(string message);
}
```

在 BookDetailViewModel 中，把 IMessageService 注入构造函数中，用于 OnSaveAsync 方法。在出现异常时调用 ShowMessageAsync 方法(代码文件 BooksLib/ViewModels/BookDetailViewModel.cs)：

```
public override async Task OnSaveAsync()
{
  try
  {
    await _itemsService.AddOrUpdateAsync(EditItem);
  }
  catch (Exception ex)
  {
    _logger.LogError("error {0} in {1}", ex.Message, nameof(OnSaveAsync));
    await _messageService.ShowMessageAsync("Error saving the data");
  }
}
```

现在只需要一个用于通用 Windows 平台的特定实现。ShowMessageAsync 方法是使用 MessageDialog 类实现的。UWPMessageService 在通用 Windows 平台应用中实现，这就是为什么现在可以访问 MessageDialog 的原因(代码文件 BooksApp/Services/UWPMessageService.cs)：

```
public class UWPMessageService : IMessageService
{
  public async Task ShowMessageAsync(string message) =>
    await new MessageDialog(message).ShowAsync();
}
```

在 ApplicationServices 构造函数中，服务通过依赖注入容器注册，UWPMessageService 类注册为 IMessageService 协定的实现类。这就是为什么视图模型接收 UWP 实现的原因(代码文件 BooksApp/ApplicationServices.cs)：

```
private ApplicationServices()
{
  var services = new ServiceCollection();
  services.AddSingleton<IBooksRepository, BooksSampleRepository>();
  services.AddSingleton<IItemsService<Book>, BooksService>();
  services.AddTransient<BooksViewModel>();
  services.AddTransient<BookDetailViewModel>();
  services.AddTransient<MainPageViewModel>();
  services.AddSingleton<IMessageService, UWPMessageService>();
  services.AddSingleton<INavigationService, UWPNavigationService>();
  services.AddSingleton<UPInitializeNavigationService>();
  services.AddLogging(builder =>
  {
#if DEBUG
    builder.AddDebug();
#endif
  });
  ServiceProvider = services.BuildServiceProvider();
}
```

如果出现错误，则可以看到如图 34-10 所示的对话框。

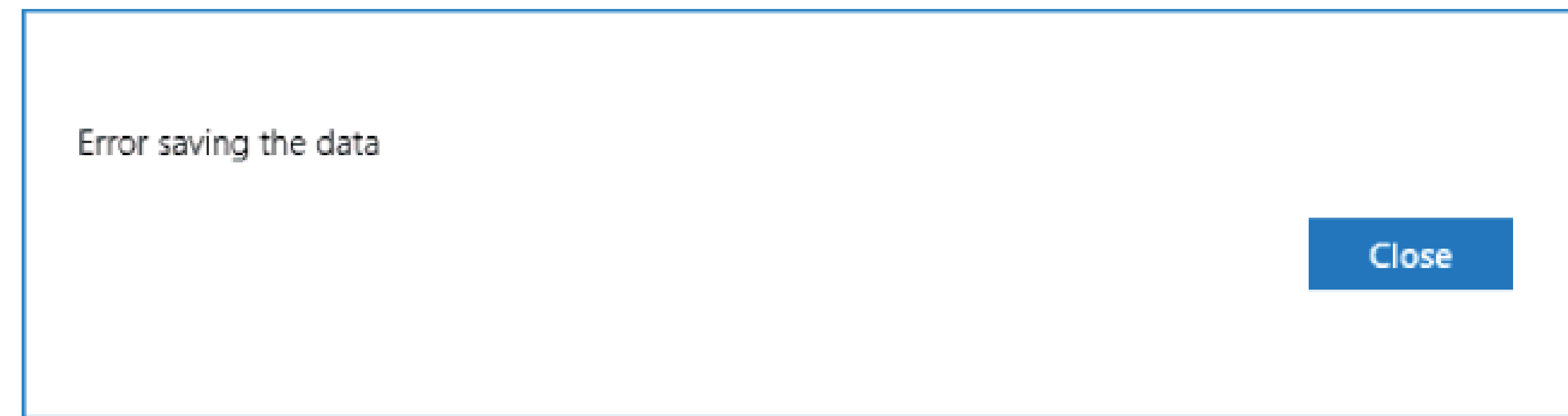



图 34-10

### 34.8.2　页面之间的导航

与打开对话框一样，不同技术之间的页面导航也不同。在 UWP 中，Frame 类用于在应用程序中导航页面。在 WPF 中，它也是一个 Frame 类，但它是一个不同的类。在 Xamarin.Forms 中， NavigationPage 用于导航。使用这些技术实现导航的方式也有所不同。在 UWP 中，需要 Type 对象来导航。在 Xamarin.Forms,中，需要页面的对象实例，其导航方法是异步的，而它们与 UWP 是同步的。为此，再次需要一个共同的协定。

在示例应用程序中，需要导航到页面，还需要进行导航回滚。此外，需要访问当前页面，以了解是否需要进行导航。为此，定义了接口 INavigationService。这个接口是基于字符串的导航，因此可以为不同的平台创建实现代码(代码文件 Framework/Services/INavigationService.cs)：

```
public interface INavigationService
{
  bool UseNavigation { get; set; }
  Task NavigateToAsync(string page);
  Task GoBackAsync();
  string CurrentPage { get; }
}
```

UWPNavigationService 需要分配一个 Frame，以便为 UWP 导航。当定义 Frame 的属性时，不可能访问它，因为在外部只使用 INavigationService 接口。在 INavigationService 接口中，不能使用 Frame，因为这个接口位于.NET 标准库中。使用 UWP 中的 INavigationService 时，可以将其转换为 UWPNavigationService，以访问特定的 UWP 属性。需要进行类型转换不是良好的设计。相反，可以轻松地指定一个特定于 UWP 的服务，比如 UWPInitializeNavigationService 并将其注入 UWPNavigationService 的构造函数中。在内部，当访问 Pages 和 Frame 属性时，这些信息从初始化服务中获取，如下面的代码片段所示(代码文件 BooksApp/Services/UWPNavigationService.cs)：

```
public class UWPNavigationService : INavigationService
{
  private readonly UWPInitializeNavigationService _initializeNavigation;

  public UWPNavigationService(
    UWPInitializeNavigationService initializeNavigation)
  {
    _initializeNavigation = initializeNavigation ??
      throw new ArgumentNullException(nameof(initializeNavigation));
  }

  private Dictionary<string, Type> _pages;
  private Dictionary<string, Type> Pages => _pages ??
    (_pages = _initializeNavigation.Pages);

  private Frame _frame;
  private Frame Frame => _frame ?? (_frame = _initializeNavigation.Frame);
  //...
}
```

CurrentPage 属性、GoBackAsync 方法和 NavigateToAsync 方法的实现现在都在使用 Frame.GoBack 和 Frame.Navigate 方法(代码文件 BooksApp/Services/UWPNavigationService.cs)：

```
public class UWPNavigationService : INavigationService
{
  //...
  private string _currentPage;
  public string CurrentPage => _currentPage;
```

```
    public Task GoBackAsync()
    {
      PageStackEntry stackEntry = Frame.BackStack.Last();
      Type backPageType = stackEntry.SourcePageType;
      KeyValuePair<string, Type> pageEntry =
        Pages.FirstOrDefault(pair => pair.Value == backPageType);
      _currentPage = pageEntry.Key;

      Frame.GoBack();
      return Task.CompletedTask;
    }

    public Task NavigateToAsync(string pageName)
    {
      _currentPage = pageName;
      Frame.Navigate(Pages[pageName]);
      return Task.CompletedTask;
    }
  }
```

UWPInitializeNavigationService 提供的唯一功能是用 Frame 和页面字典初始化它，并检索这个信息(代码文件 BooksApp/Services/UWPInitializeNavigationService.cs)：

```
  public class UWPInitializeNavigationService
  {
    public void Initialize(Frame frame, Dictionary<string, Type> pages)
    {
      Frame = frame ?? throw new ArgumentNullException(nameof(frame));
      Pages = pages ?? throw new ArgumentNullException(nameof(pages));
    }
    public Frame Frame { get; private set; }
    public Dictionary<string, Type> Pages { get; private set; }
  }
```

现在可以在 Frame 可用的位置初始化 UWPInitializeNavigationService。在 UWP 示例应用程序中，这个位置在 MainPage 中。在前面指定的 NavigationView 控件中，指定名为 ContentFrame 的框架。现在可以在 MainPage 的代码隐藏文件中或特定于 UWP 的 MainPageViewModel 中定义初始化。对于示例应用程序，选择第二个选项。

在下面的代码片段中，MainPageViewModel 保存了用于导航的页面列表，并在调用 SetNavigationFrame 时初始化导航服务(代码文件 BooksApp/ViewModels/MainPageViewModel.cs)：

```
  public class MainPageViewModel : ViewModelBase
  {
    private Dictionary<string, Type> _pages = new Dictionary<string, Type>
    {
      [PageNames.BooksPage] = typeof(BooksPage),
      [PageNames.BookDetailPage] = typeof(BookDetailPage)
    };

    private readonly INavigationService _navigationService;
    private readonly UWPInitializeNavigationService _initializeNavigationService;
    public MainPageViewModel(INavigationService navigationService,
      UWPInitializeNavigationService initializeNavigationService)
    {
      _navigationService = navigationService;
      _initializeNavigationService = initializeNavigationService;
    }

    public void SetNavigationFrame(Frame frame) =>
      _initializeNavigationService.Initialize(frame, _pages);

    //...
  }
```

有了这个视图模型，MainPage 的代码隐藏文件中需要 ViewModel 属性，并通过调用 SetNavigationFrame 方法将 ContentFrame 传递给导航服务 (代码文件 BooksApp/MainPage.xaml.cs)：

```
  public sealed partial class MainPage : Page
  {
    public MainPage()
    {
      this.InitializeComponent();
      ViewModel =
        (Application.Current as App).AppServices.GetService<MainPageViewModel>();
      ViewModel.SetNavigationFrame(ContentFrame);
```

```
  }

  public MainPageViewModel ViewModel { get; }
}
```

BooksPage 的第一个导航发生在 MainPageViewModel 中。方法 OnNavigationSelectionChanged 是 NavigationView 控件的 NavigationSelectionChanged 事件处理程序。把 Tag 设置为 books，使用 INavigationService 导航到 BooksPage (代码文件 BooksApp/ViewModels/MainPageViewModel.cs)：

```
public class MainPageViewModel : ViewModelBase
{
  //...
  public void OnNavigationSelectionChanged(NavigationView sender,
    NavigationViewSelectionChangedEventArgs args)
  {
    if (args.SelectedItem is NavigationViewItem navigationItem)
    {
      switch (navigationItem.Tag)
      {
        case "books":
          _navigationService.NavigateToAsync(PageNames.BooksPage);
          break;
        default:
          break;
      }
    }
  }
}
```

从 BooksPage 的导航直接在共享的视图模型中执行。从 BooksPage 到 BooksDetailPage 的导航在选择列表项时发生，因此触发 PropertyChanged 事件。导航也只有在 UseNavigation 属性设置为 true 时才能完成。如前所述，在 UWP 中，当 UI 足够大时，在这个地方不需要导航，因为详细信息会与列表并排显示(代码文件 BooksLib/ViewModels/BooksViewModel.cs)：

```
public class BooksViewModel : MasterDetailViewModel<BookItemViewModel, Book>
{
  private readonly IItemsService<Book> _booksService;
  private readonly INavigationService _navigationService;

  public BooksViewModel(IItemsService<Book> booksService,
    INavigationService navigationService)
    : base(booksService)
  {
    _booksService = booksService ??
      throw new ArgumentNullException(nameof(booksService));
    _navigationService = navigationService ??
      throw new ArgumentNullException(nameof(navigationService));

    PropertyChanged += async (sender, e) =>
    {
      if (UseNavigation && e.PropertyName == nameof(SelectedItem) &&
        _navigationService.CurrentPage == PageNames.BooksPage)
      {
        await _navigationService.NavigateToAsync(PageNames.BookDetailPage);
      }
    };
  }

  public bool UseNavigation { get; set; }
  //...
}
```

为了使用户界面小到需要在页面之间进行导航，下一节将解释自适应用户界面。

### 34.8.3　自适应用户界面

根据屏幕上可用的空间，用户控件应该在列表控件或单独的页面中并排显示。为此，通过 BookDetailUserControl

在 BooksPage 中使用，Visibility 属性默认设置为 Collapsed(代码文件 BooksApp/Views/BooksPage.xaml)：

```
<views:BookDetailUserControl x:Name="BookDetailUC" Visibility="Collapsed"
  Grid.Column="1" Grid.Row="1" Grid.RowSpan="2" />
```

现在，可以使用 AdaptiveTrigger 来使控件显示在宽度大于 1023 的窗口中。当 AdaptiveTrigger 触发时，Setter 将用户控件的 Visibility 属性设置为 Visible(代码文件 BooksApp/Views/BooksPage.xaml)：

```
<Grid>
  <!-- ... -->
  <views:BookDetailUserControl x:Name="BookDetailUC" Visibility="Collapsed"
    Grid.Column="1" Grid.Row="1" Grid.RowSpan="2" />

  <VisualStateManager.VisualStateGroups>
    <VisualStateGroup x:Name="WindowStates">
      <VisualState x:Name="WideState">
        <VisualState.StateTriggers>
          <AdaptiveTrigger MinWindowWidth="1024"/>
        </VisualState.StateTriggers>
        <VisualState.Setters>
          <Setter Target="BookDetailUC.Visibility" Value="Visible" />
        </VisualState.Setters>
      </VisualState>
      <VisualState x:Name="MediumState">
        <VisualState.StateTriggers>
          <AdaptiveTrigger MinWindowWidth="720"/>
        </VisualState.StateTriggers>
      </VisualState>
      <!-- ... -->
    </VisualStateGroup>
  </VisualStateManager.VisualStateGroups>
</Grid>
```

运行该应用程序时，可以看到显示用户控件的 BooksPage(参见图 34-11)。图 34-12 显示了隐藏用户控件的较小状态。在这里，NavigationView 也会引发一个更小的视图。使应用程序更小，设置为 NarrowState(参见图 34-13)，带有 NavigationView 的 AdaptiveTrigger 会再次触发，只显示汉堡包按钮。然后，只有单击汉堡包按钮，才显示 NavigationView 的窗格。

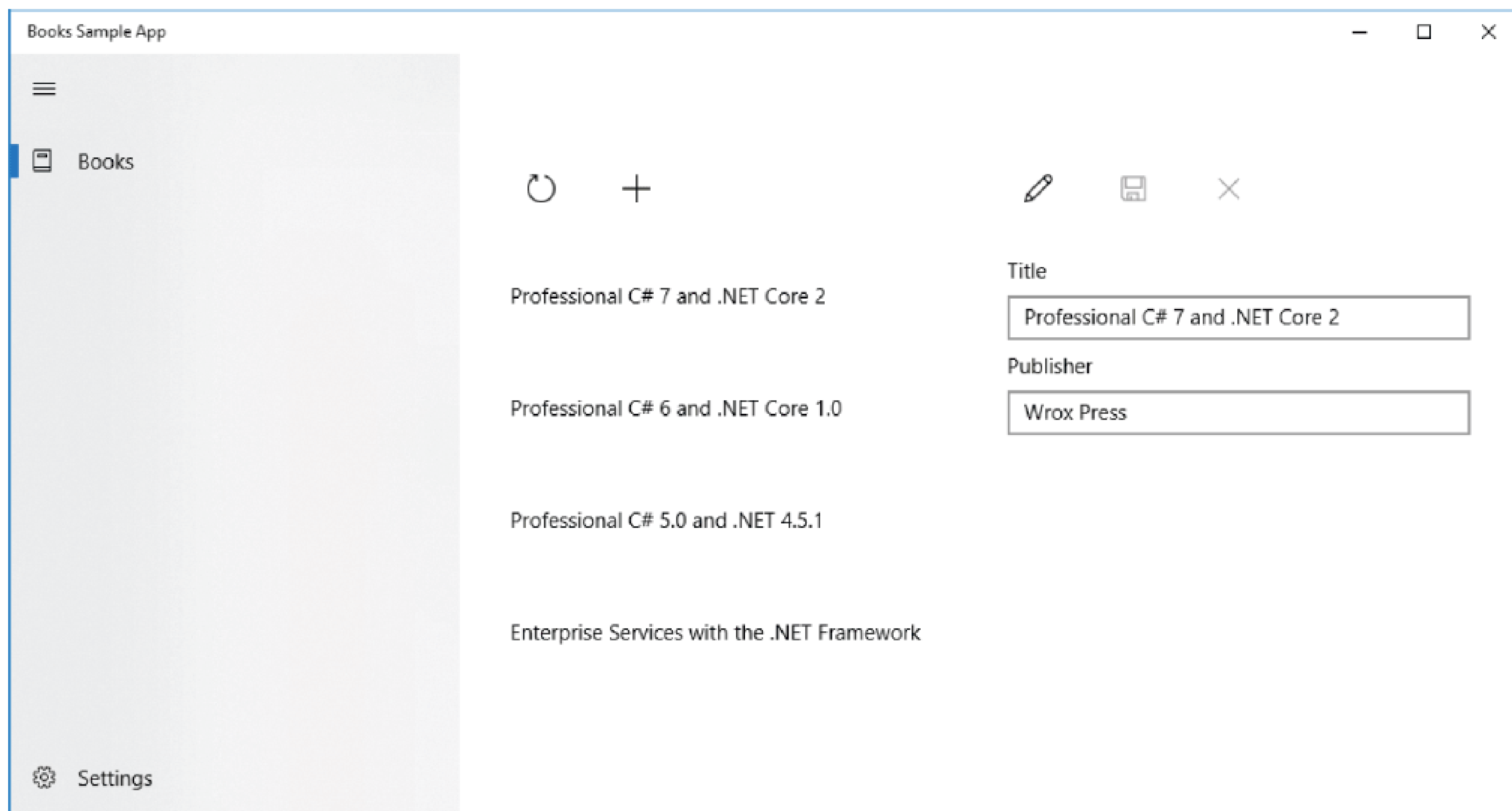



图 34-11

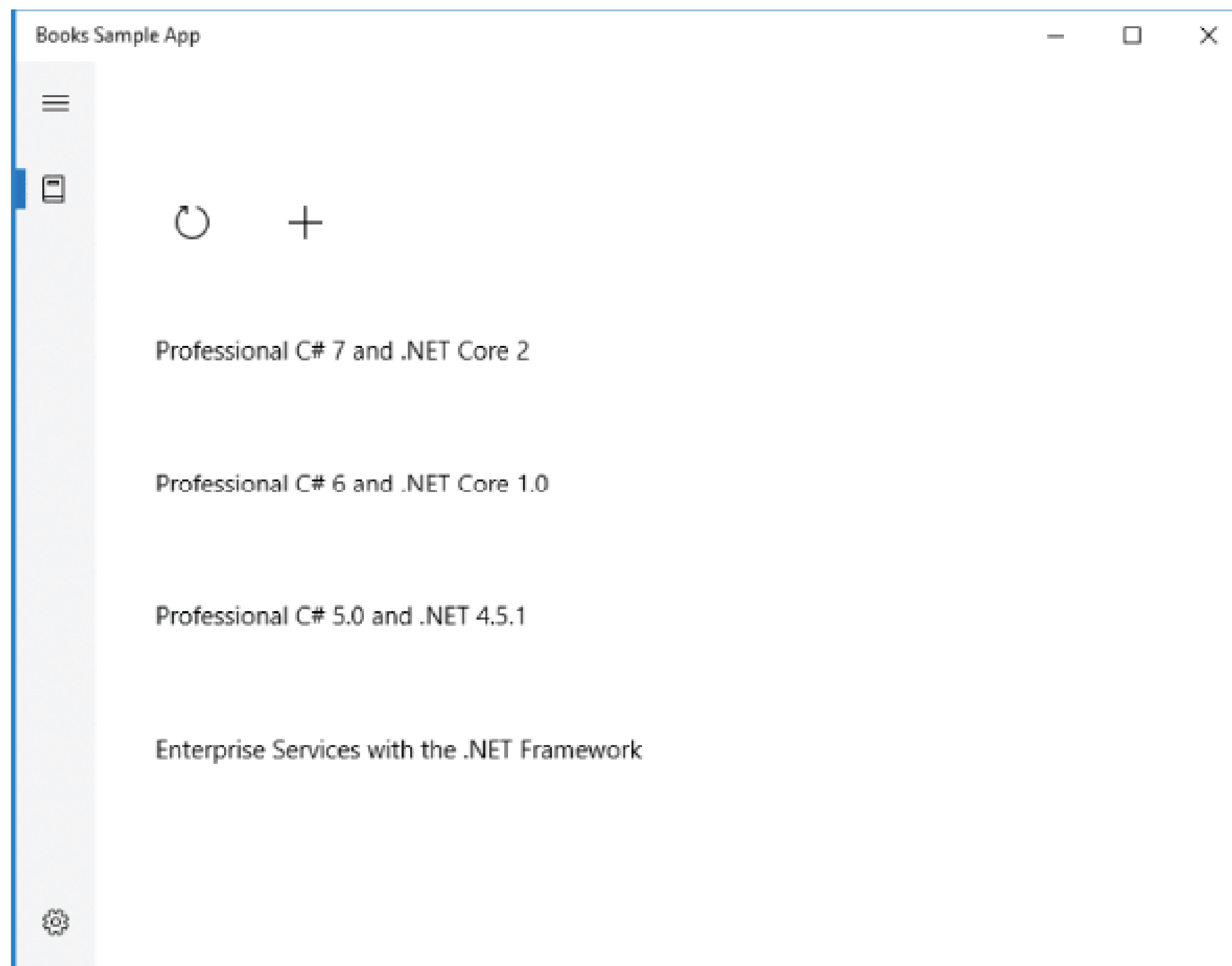



图 34-12

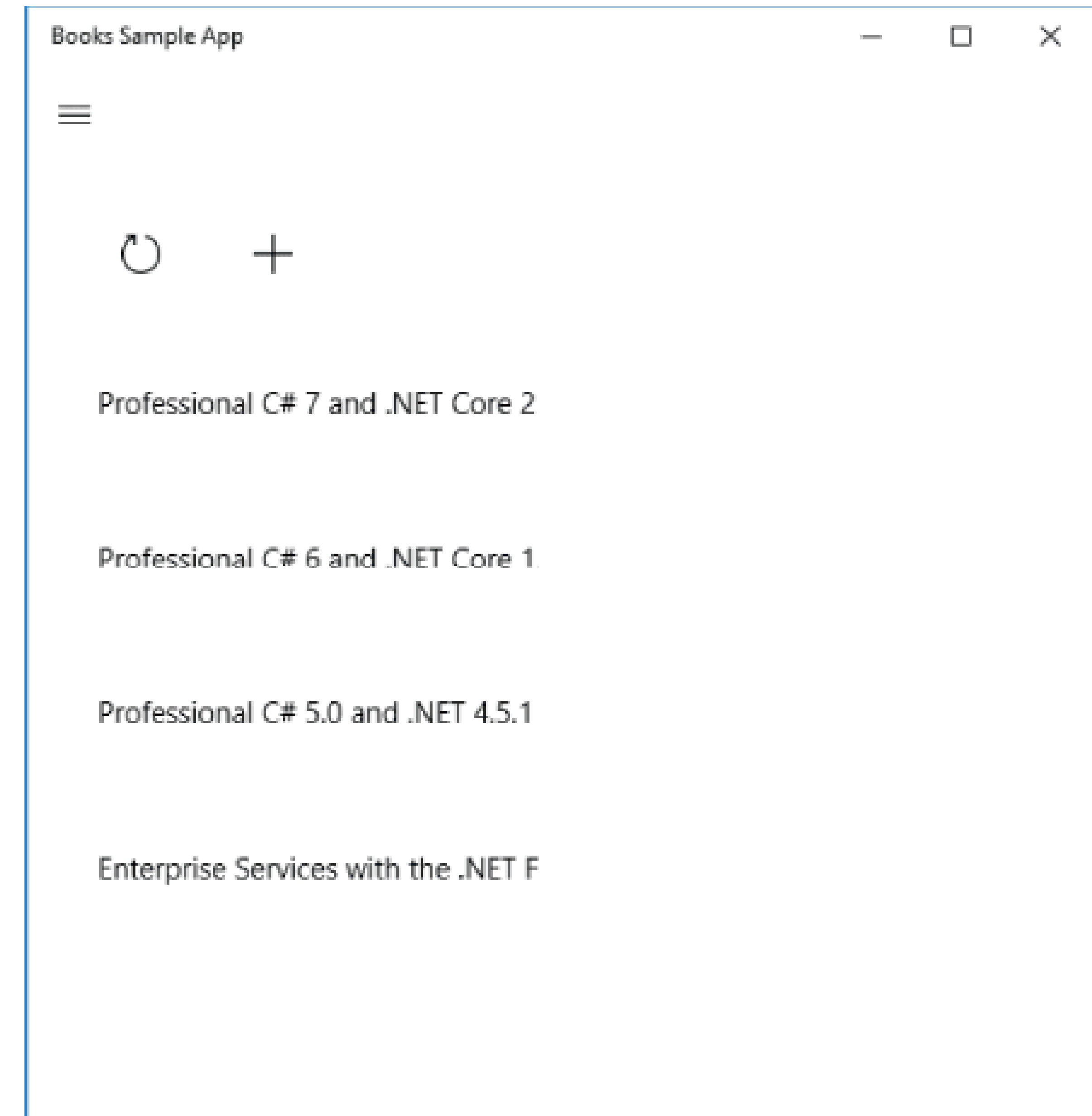



图 34-13

**注意：**
有关自适应触发器的更多信息，请参阅第 33 章。

**注意：**
这里缺少的是更改视图模型的 UseNavigation 属性，以使用或不使用导航。一种方法是直接或通过服务设置视图模型的属性。另一种方法是使用事件聚合器，参见本章 34.8.6 节“使用事件传递消息”。

## 34.8.4 显示进度信息

在许多应用程序和许多视图模型中，显示进度信息是必要的。这就是为什么在示例应用程序中，在 ViewModelBase 类中实现了进展信息的显示。

在视图模型中显示进度信息可以像布尔显示属性 ShowProgress 一样简单。对于可以启动多个并行操作的更复杂场景，可以使用计数器创建实现代码。在调用 SetInProgress 方法之后，为了不忘记再次减少计数，StartInProgress 方法返回一个 IDisposable，它会自动对 Dispose 进行递减(代码文件 Framework/ViewModels/ViewModelBase.cs)：

```
public abstract class ViewModelBase : BindableBase
{
  private class StateSetter : IDisposable
  {
    private Action _end;
    public StateSetter(Action start, Action end)
    {
      start?.Invoke();
      _end = end;
    }
    public void Dispose() => _end?.Invoke();
  }

  private int _inProgressCounter = 0;
  protected void SetInProgress(bool set = true)
  {
    if (set)
    {
      Interlocked.Increment(ref _inProgressCounter);
      OnPropertyChanged(nameof(InProgress));
    }
    else
    {
      Interlocked.Decrement(ref _inProgressCounter);
```

```
          OnPropertyChanged(nameof(InProgress));
        }
      }

      public IDisposable StartInProgress() =>
        new StateSetter(() => SetInProgress(), () => SetInProgress(false));

      public bool InProgress => _inProgressCounter != 0;
      //...
    }
```

对于每个可能需要更长时间的动作，例如，用于保存的 EndEdit 方法，调用 StartInProgress 方法，将该方法返回的内容传递给 using 语句。因此，在 using 语句的末尾调用 Dispose 方法，再次减少进度计数(代码文件 Framework/ViewModels/EditableViewModel.cs)：

```
public async virtual void EndEdit()
{
  using (StartInProgress())
  {
    await OnSaveAsync();
    EditItem = default(TItem);
    IsEditMode = false;
    await _itemsService.RefreshAsync();
    await OnEndEditAsync();
  }
}
```

现在，可以使用 ProgressBar，将 Visibility 属性绑定到视图模型的 InProgress 属性，并自动显示进度条(代码文件 BooksApp/Views/BooksPage.xaml)：

```
<ProgressBar Margin="8" HorizontalAlignment="Stretch"
  Visibility="{x:Bind ViewModel.InProgress, Mode=OneWay}"
  IsIndeterminate="True" Grid.ColumnSpan="2" />
```

### 34.8.5 使用列表项中的操作

在现代用户界面中，当显示列表时，列表项还提供了一些附加信息，或者允许在项上悬停时执行一些操作。例如，微软的 Mail 程序允许用户在不首先选择邮件的情况下就可以标记/删除/归档邮件。这是怎么做到的？

可以绑定 BookItemViewModel 对象，而不是直接将 Book 项目绑定到列表中。使用 BookItemViewModel 项，除了图书的信息之外，还可以定义命令。

在样例代码中，BookItemViewModel 是包装 Book 对象的视图模型。与其他视图模型相反，它不会通过 DI 容器创建，因此可以在构造函数中使用 Book 对象。book 实例分配给 ItemViewModel 基类的 Item 属性。除了基类之外，BookItemViewModel 还定义了 DeleteBookCommand 和 OnDeleteBook 方法，该方法调用 IItemsService 的 DeleteAsync 方法(代码文件 BooksLib/ViewModels/BookItemViewModel.cs)：

```
public class BookItemViewModel : ItemViewModel<Book>
{
  private readonly IItemsService<Book> _booksService;

  public BookItemViewModel(Book book, IItemsService<Book> booksService)
  {
    Item = book;
    _booksService = booksService;
    DeleteBookCommand = new RelayCommand(OnDeleteBook);
  }

  public RelayCommand DeleteBookCommand { get; set; }

  private async void OnDeleteBook()
  {
    await _booksService.DeleteAsync(Item);
  }
}
```

当 BookItemViewModel 没有通过 DI 容器创建时，它是如何创建的呢？MasterDetailViewModel 不仅定义了可以绑定到列表控件的 Items 属性，还定义了 ItemsViewModels 属性。这个属性只使用抽象方法 ToViewModel 将 Items 转换为 TItemViewModel(代码文件 Framework/ViewModels/MasterDetailViewModel.cs)：

```
public abstract class MasterDetailViewModel<TItemViewModel, TItem> :
  ViewModelBase
  where TItemViewModel : IItemViewModel<TItem>
  where TItem: class
{
  //...
  public ObservableCollection<TItem> Items => _itemsService.Items;

  protected abstract TItemViewModel ToViewModel(TItem item);

  public virtual IEnumerable<TItemViewModel> ItemsViewModels =>
    Items.Select(item => ToViewModel(item));
  //...
}
```

派生类 BooksViewModel 只需要重写 ToViewModel 方法，以将 Book 对象转换为 BookItemViewModel 对象。为此，只需要创建 BookItemViewModel 对象，来传递 Book 和所需的服务(代码文件 BooksLib/ViewModels/BooksViewModel.cs)：

```
protected override BookItemViewModel ToViewModel(Book item) =>
  new BookItemViewModel(item, _booksService);
```

现在可以将视图模型类型绑定到 ListView 上。定义一个用户控件，以显示列表项。如何确保只有当鼠标悬停在列表项时，才显示按钮？已经在自适应应用户界面中用过的一种技术是，把一个元素(本例中是 Grid 控件) 的 visibility 设置为 Collapsed。现在，使用 HoverButtonsShown 和 HoverButtonsHidden 定义定制的 Visual State 元素。在状态 HoverButtonsShown 中，Grid 的 visibility 设置为 Visible(代码文件 BooksApp/Views/BookItemUserControl.xaml)：

```
<Grid>
  <VisualStateManager.VisualStateGroups>
    <VisualStateGroup x:Name="HoveringStates">
      <VisualState x:Name="HoverButtonsShown">
        <VisualState.Setters>
          <Setter Target="hoverArea.Visibility" Value="Visible" />
        </VisualState.Setters>
      </VisualState>
      <VisualState x:Name="HoverButtonsHidden">
      </VisualState>
    </VisualStateGroup>
  </VisualStateManager.VisualStateGroups>

  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="*" />
    <ColumnDefinition Width="auto" />
  </Grid.ColumnDefinitions>
  <StackPanel VerticalAlignment="Center">
    <TextBlock
      Text="{x:Bind Mode=OneWay, Path=BookItemViewModel.Item.Title}" />
  </StackPanel>
  <Grid Grid.Column="1" x:Name="hoverArea" Visibility="Collapsed"
    VerticalAlignment="Center">
    <AppBarButton
      Command="{x:Bind Mode=OneWay, Path=BookItemViewModel.DeleteBookCommand}"
      Icon="Delete" Label="Delete" IsTabStop="False"
      VerticalAlignment="Stretch"/>
  </Grid>
</Grid>
```

在代码隐藏文件中，通过 OnPointerEntered 和 OnPointerExited,重写方法，使用 VisualStateManager 设置状态。只有当用户使用鼠标或钢笔时，才设置状态(代码文件 BooksApp/Views/BookItemUserControl.xaml.cs)：

```
public sealed partial class BookItemUserControl : UserControl
{
  public BookItemUserControl()
  {
    this.InitializeComponent();
  }

  public BookItemViewModel BookItemViewModel
  {
    get => (BookItemViewModel)GetValue(BookItemViewModelProperty);
    set => SetValue(BookItemViewModelProperty, value);
  }
```

```
    public static readonly DependencyProperty BookItemViewModelProperty =
      DependencyProperty.Register("BookViewModel", typeof(BookItemViewModel),
      typeof(BookItemUserControl), new PropertyMetadata(null));

    protected override void OnPointerEntered(PointerRoutedEventArgs e)
    {
      base.OnPointerEntered(e);

      if (e.Pointer.PointerDeviceType == PointerDeviceType.Mouse ||
          e.Pointer.PointerDeviceType == PointerDeviceType.Pen)
      {
        VisualStateManager.GoToState(this, "HoverButtonsShown", true);
      }
    }

    protected override void OnPointerExited(PointerRoutedEventArgs e)
    {
      base.OnPointerExited(e);

      VisualStateManager.GoToState(this, "HoverButtonsHidden", true);
    }
  }
```

运行应用程序时，把鼠标悬停在列表项上时，就可以看到状态的变化，如图 34-14 所示。

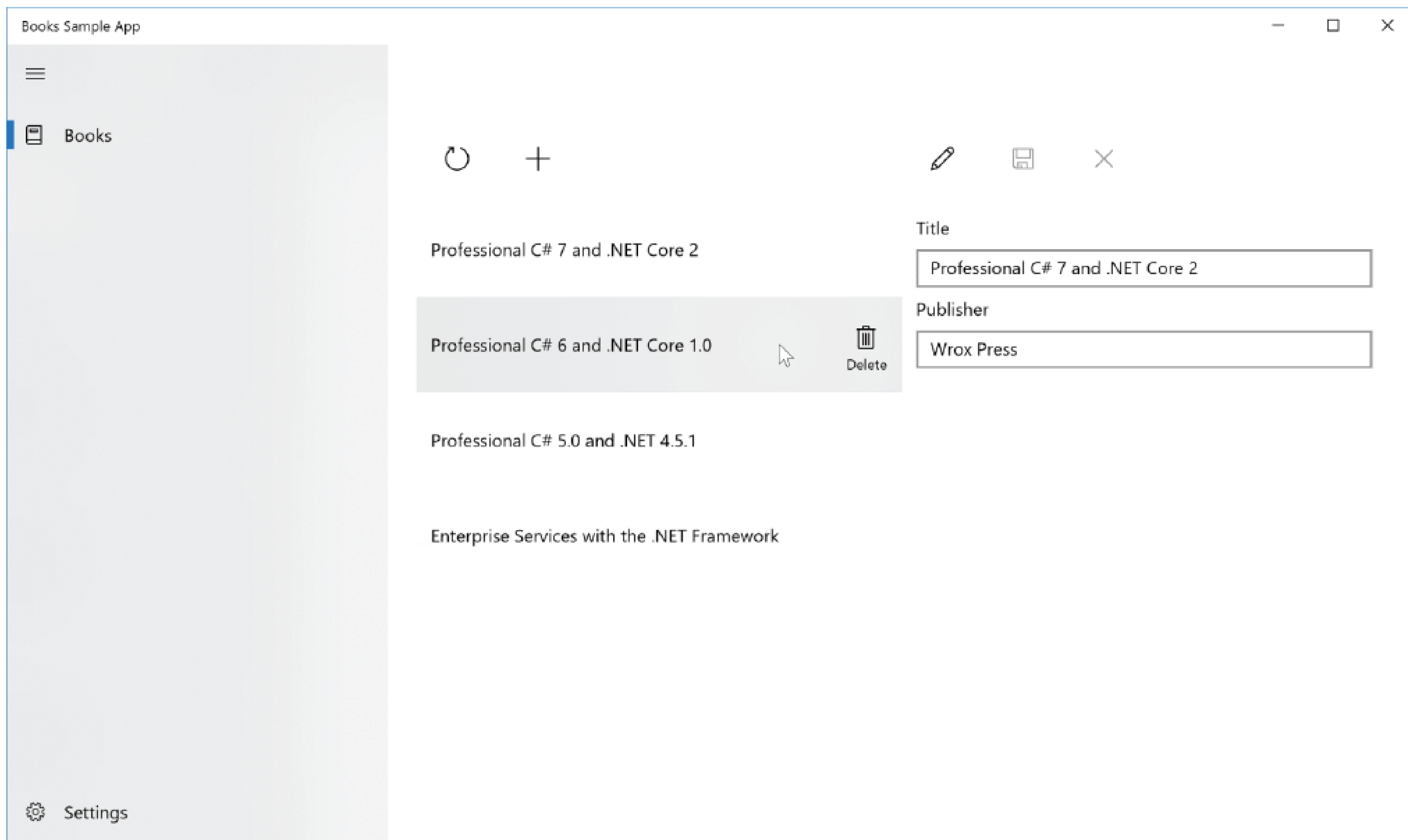



图 34-14

## 34.9 使用事件传递消息

对于应用程序的当前状态，有一个问题需要根据应用程序窗口的大小，设置视图模型的状态，以使用导航。处理此问题的一种方法是创建带有事件的单例服务，使用视图模型中的这个服务来订阅事件，然后主窗口触发事件。这样的服务也可以在全球范围内定义——为事件定义的服务。对于许多 MVVM 框架，这也称为事件聚合器。

Framework 项目定义了一个泛型 EventAggregator。该聚合器定义了一个名为 Event 的事件，其中 Action <object, TEvent>类型的处理程序可以订阅和取消订阅，方法 Publish 触发事件。这个聚合器实现为一个单例，使其易于访问，而不需要创建实例(代码文件 Framework/EventAggregator.cs)：

```
public class EventAggregator<TEvent>
  where TEvent: EventArgs
```

```
{
  private static EventAggregator<TEvent> s_eventAggregator;

  public static EventAggregator<TEvent> Instance =>
    s_eventAggregator ?? (s_eventAggregator = new EventAggregator<TEvent>());

  private EventAggregator()
  {
  }

  public event Action<object, TEvent> Event;

  public void Publish(object source, TEvent ev) =>
    Event?.Invoke(source, ev);
}
```

**注意：**

对于泛型的单例类，不止创建一个实例，每个泛型参数类型都有一个实例。这对于 EventAggregator 来说很好，因为不同的事件类型不需要共享一些数据，并且它允许更好的可伸缩性。

要将有关导航的信息从主窗口传递到视图模型，需要使用 NavigationInfoEvent。这个事件信息类使用布尔属性来定义是否应该使用导航(代码文件 BooksLib/Events/NavigationInfoEvent.cs)：

```
public class NavigationInfoEvent : EventArgs
{
  public bool UseNavigation { get; set; }
}
```

BooksViewModel 现在可以订阅事件。使用 BooksViewModel 的构造函数可以访问静态成员 Instance，以获取 NavigationInfoEvent 类型的 singleton 对象，并将 UseNavigation 属性方法分配给事件信息 e.UseNavigation(代码文件 BooksLib/ViewModels/BooksViewModel.cs)：

```
public BooksViewModel(IItemsService<Book> booksService,
  INavigationService navigationService)
  : base(booksService)
{
  _booksService = booksService ??
    throw new ArgumentNullException(nameof(booksService));
  _navigationService = navigationService ??
    throw new ArgumentNullException(nameof(navigationService));

  EventAggregator<NavigationInfoEvent>.Instance.Event += (sender, e) =>
  {
    UseNavigation = e.UseNavigation;
  };
  //...
}
```

当主页面的大小发生变化时，将发布该事件。在 MainPage 类中，OnSizeChanged 事件处理程序注册到页面的 SizeChanged 事件。在事件处理程序中，EventAggregator 可以像使用静态 Instance 属性来访问订阅那样，通过调用 Publish 方法来访问，该方法传递一个 NavigationInfoEvent 对象。NavigationInfoEvent 对象根据窗口的大小进行初始化(代码文件 ViewModels/BooksViewModel.cs)：

```
public sealed partial class MainPage : Page
{
  //...
  private void OnSizeChanged(object sender, SizeChangedEventArgs e)
  {
    EventAggregator<NavigationInfoEvent>.Instance.Publish(this,
      new NavigationInfoEvent { UseNavigation = e.NewSize.Width < 1024 });
  }
}
```

## 34.10　使用框架

在示例应用程序中，看到 Framework 项目中定义的类，例如，BindableBase、DelegateCommand 和

EventAggregator。基于 MVVM 的应用程序需要这些类，但不需要自己来实现它们。其工作量不大，但可以使用现有的 MVVM 框架。

Laurent Bugnion 的 MVVM Light(http://mvvmlight.net)是一个小框架，完全符合 MVVM 应用程序的目的，可用于许多不同的平台。

另一个框架 Prism.Core ( http://github.com/PrismLibrary )最初由 Microsoft Patterns and Practices 团队创建，现在转移到社区。虽然 Prism 框架非常成熟，支持插件和定位控件的区域，但 Prism.Core 很轻，仅包含几个类型，如 BindableBase、DelegateCommand 和 ErrorsContainer。

## 34.11 小结

本章围绕 MVVM 模式提供了创建基于 XAML 的应用程序的架构指南。讨论了模型、视图和视图模型的关注点分离。除此之外，还介绍了使用接口 INotifyPropertyChanged 实现更改通知，分离数据访问代码的存储库模式，使用事件在视图模型之间传递消息(这也可以用来与视图通信)，以及使用 IoC 容器注入依赖项。

本章还展示了可以跨应用程序使用的视图模型库。这些都允许代码共享，同时仍然允许使用特定平台的功能。可以通过库和服务实现使用特定于平台的特征，协定可用于所有的平台。第 37 章利用同样的库创建了一个带有 Xamarin 的应用程序。

第 35 章将继续讨论 XAML、样式和资源。

# 第35章

# 样式化 Windows 应用程序

**本章要点**

- 为 Windows 应用程序指定样式
- 用形状和几何形状创建基础图
- 用转换进行缩放、旋转和扭曲
- 使用笔刷填充背景
- 处理样式、模板和资源
- 创建动画
- Visual State Manager

**本章源代码下载地址(wrox.com)：**

打开 www.wrox.com 的 Download Code 选项卡可下载本章源代码。源代码也可以在 Styles 目录的 https://github.com/ProfessionalCSharp/ProfessionalCSharp7 中找到。本章代码分为以下几个主要的示例文件：

- Shapes
- Geometry
- Transformation
- Brushes
- Styles and Resources
- Template
- Animation
- Transitions
- Visual State

## 35.1 样式设置

近年来，开发人员越来越关心应用程序的外观。当 Windows Forms 是创建桌面应用程序的技术时，用户界面没有提供许多设置应用程序样式的选项。控件有标准的外观，根据正在运行应用程序的操作系统版本而略有不同，但不大容易定义完整自定义的外观。

Windows Presentation Foundation(WPF)改变了这一切。WPF 基于 DirectX，从而提供了向量图形，允许方便地调整窗口和控件的大小。控件是完全可定制的，可以有不同的外观。设置应用程序的样式变得非常重要。应用程序可以有任何外观。有了优秀的设计，用户可以使用应用程序，而不需要知道如何使用 Windows 应用程序。相反，用户只需要拥有特定领域的知识。例如，苏黎世机场创建了一个 WPF 应用程序，其中的按钮看起来像飞机。通过按钮，用户可以获取飞机的位置信息(完整的应用程序看起来像机场)。按钮的颜色可以根据配置有不同的含义：它们可以显示航线或飞机的准时/延迟信息。通过这种方式，应用程序的用户很容易看到目前在机场的飞机有或长或短的延误。

应用程序拥有不同的外观，这对于现代 Windows 应用程序更加重要。在这些应用程序中，以前没有使用过 Windows 应用程序的用户可以使用这些设备。对于非常熟悉 Windows 应用程序的用户，应该考虑通过使用户工作得更方便的典型过程，帮助这些用户提高效率。

在设置 WPF 应用程序的样式时，微软公司没有提供很多指导。应用程序的外观主要取决于设计人员的想象力。对于 UWP 应用程序，微软公司提供了更多的指导和预定义的样式，也能够修改任意样式。

本章首先介绍 XAML 的核心元素 shapes，它允许绘制线条、椭圆和路径元素。介绍了形状的基础之后，就讨论 geometry 元素。可以使用 geometry 元素来快速创建基于矢量的图形。

使用 transformation，可以缩放、旋转任何 XAML 元素。用 brush 可以创建纯色、渐变或更高级的背景。本章将论述如何在样式中使用画笔和把样式放在 XAML 资源中。

最后，使用 template 模板可以完全自定义控件的外观，本章还要学习如何创建动画。

## 35.2　形状

形状是 XAML 的核心元素。利用形状，可以绘制矩形、线条、椭圆、路径、多边形和折线等二维图形，这些图形用派生自抽象类 Shape 的类表示。图形在 Windows. UI.Xaml.Shapes 名称空间中定义。

下面的 XAML 示例绘制了一个黄色笑脸，它用一个椭圆表示笑脸，两个椭圆表示眼睛，两个椭圆表示眼睛中的瞳孔，一条路径表示嘴型(代码文件 Shapes/MainPage.xaml)：

```
<Canvas>
  <Ellipse Canvas.Left="10" Canvas.Top="10" Width="100" Height="100"
    Stroke="Blue" StrokeThickness="4" Fill="Yellow" />
  <Ellipse Canvas.Left="30" Canvas.Top="12" Width="60" Height="30">
    <Ellipse.Fill>
      <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5, 1">
        <GradientStop Offset="0.1" Color="DarkGreen" />
        <GradientStop Offset="0.7" Color="Transparent" />
      </LinearGradientBrush>
    </Ellipse.Fill>
  </Ellipse>
  <Ellipse Canvas.Left="30" Canvas.Top="35" Width="25" Height="20"
    Stroke="Blue" StrokeThickness="3" Fill="White" />
  <Ellipse Canvas.Left="40" Canvas.Top="43" Width="6" Height="5"
    Fill="Black" />
  <Ellipse Canvas.Left="65" Canvas.Top="35" Width="25" Height="20"
    Stroke="Blue" StrokeThickness="3" Fill="White" />
  <Ellipse Canvas.Left="75" Canvas.Top="43" Width="6" Height="5"
    Fill="Black" />
  <Path Stroke="Blue" StrokeThickness="4"
    Data="M 40,74 Q 57,95 80,74" />
</Canvas>
```

图 35-1 显示了这些 XAML 代码的结果。

无论是按钮还是线条、矩形等图形，所有这些 XAML 元素都可以通过编程来访问。把 Path 元素的 Name 或 x:Name 属性设置为 mouth，就可以用变量名 mouth 以编程方式访问这个元素：

```
<Path Name="mouth" Stroke="Blue" StrokeThickness="4"
  Data="M 40,74 Q 57,95 80,74 " />
```



图 35-1

接下来更改代码，脸上的嘴在后台代码中动态改变。添加一个按钮和单击处理程序，在其中调用 SetMouth 方法(代码文件 Shapes/MainPage.xaml.cs)：

```
private void OnChangeShape(object sender, RoutedEventArgs e)
{
  SetMouth();
}
```

在后台隐藏文件中，可以使用图片和片段创建几何图形。首先，创建一个二维数组，其中包含的 6 个点定义了表示快乐状态的 3 个点，和表示悲伤状态的 3 个点(代码文件 Shapes/MainPage. xaml.cs)：

```
private readonly Point[,] _mouthPoints = new Point[2, 3]
{
  {
    new Point(40, 74), new Point(57, 95), new Point(80, 74),
  },
  {
    new Point(40, 82), new Point(57, 65), new Point(80, 82),
  }
};
```

接下来，将一个新的 PathGeometry 对象分配给 Path 的 Data 属性。PathGeometry 包含定义了起点的 PathFigure(设置 StartPoint 属性与路径标记语法中的字母 M 是一样的)。PathFigure 包含 QuadraticBezierSegment，其中的两个 Point 对象分配给属性 Point1 和 Point2 (与带有两个点的字母 Q 一样)：

```
private bool _laugh = false;
public void SetMouth()
{
  int index = _laugh ? 0: 1;

  var figure = new PathFigure() { StartPoint = _mouthPoints[index, 0] };
  figure.Segments = new PathSegmentCollection();
  var segment1 = new QuadraticBezierSegment
  {
    Point1 = _mouthPoints[index, 1];
    Point2 = _mouthPoints[index, 2];
  }

  figure.Segments.Add(segment1);
  var geometry = new PathGeometry();
  geometry.Figures = new PathFigureCollection();
  geometry.Figures.Add(figure);

  mouth.Data = geometry;
  _laugh = !_laugh;
}
```

分段和图片的使用在下一节详细说明。运行应用程序，图 35-2 显示悲伤的脸。

表 35-1 描述了名称空间 System.Windows.Shapes 和 Windows.Ui.Xaml.Shapes 中可用的形状。



图 35-2

表 35-1

| Shape 类 | 说　明 |
|---|---|
| Line | 可以在坐标(X1,Y1)到(X2,Y2)之间绘制一条线 |
| Rectangle | 使用 Rectangle 类，通过指定 Width 和 Height 可以绘制一个矩形 |
| Ellipse | 使用 Ellipse 类，可以绘制一个椭圆 |
| Path | 使用 Path 类可以绘制一系列直线和曲线。Data 属性是 Geometry 类型。还可以使用派生自基类 Geometry 的类绘制图形，或使用路径标记语法来定义图形 |
| Polygon | 使用 Polygon 类可以绘制由线段连接而成的封闭图形。多边形由一系列赋予 Points 属性的 Point 对象定义 |
| Polyline | 类似于 Polygon 类，使用 Polyline 也可以绘制连接起来的线段。与多边形的区别是，折线不一定是封闭图形 |

## 35.3 几何图形

前面示例显示，其中一种形状 Path 使用 Geometry 来绘图。Geometry 元素也可用于其他地方，如用于 DrawingBrush。

在某些方面，Geometry 元素非常类似于形状。与 Line、Ellipse 和 Rectangle 形状一样，也有绘制这些形状的 Geometry 元素：LineGeometry、EllipseGeometry 和 RectangleGeometry。形状与几何图形有显著的区别。Shape 是一个 FrameworkElement，可以用于把 UIElement 用作其子元素的任意类。FrameworkElement 派生自 UIElement。形状会参与系统的布局，并呈现自身。而 Geometry 类不呈现自身，特性和系统开销也比 Shape 类少。Geometry 类直接派生自 DependencyObject。

Path 类使用 Geometry 来绘图。几何图形可以用 Path 的 Data 属性设置。可以设置的简单的几何图形元素有绘制椭圆的 EllipseGeometry、绘制线条的 LineGeometry 和绘制矩形的 RectangleGeometry。

### 35.3.1 使用段的几何图形

也可以使用段来创建几何图形。几何图形类 PathGeometry 使用段来绘图。下面的代码段使用 BezierSegment 和 LineSegment 元素绘制一个红色的图形和一个绿色的图形，如图 35-3 所示。第一个 BezierSegment 在图形的起点(70,40)、终点(150,63)、控制点(90,37)和(130,46)之间绘制了一条贝塞尔曲线。下面的 LineSegment 使用贝塞尔曲线的终点和(120,110)绘制了一条线段(代码文件 Geometries/MainPage.xaml )：

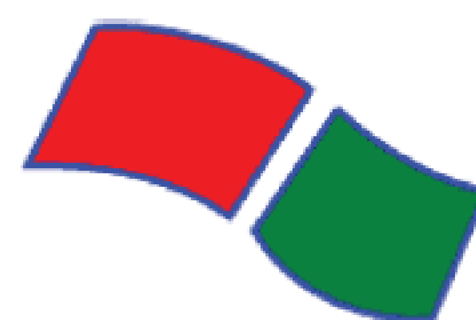


图 35-3

```
<Path Canvas.Left="0" Canvas.Top="0" Fill="Red" Stroke="Blue"
  StrokeThickness="2.5">
  <Path.Data>
    <GeometryGroup>
      <PathGeometry>
        <PathGeometry.Figures>
          <PathFigure StartPoint="70,40" IsClosed="True">
            <PathFigure.Segments>
              <BezierSegment Point1="90,37" Point2="130,46"
                Point3="150,63" />
              <LineSegment Point="120,110" />
              <BezierSegment Point1="100,95" Point2="70,90"
                Point3="45,91" />
            </PathFigure.Segments>
          </PathFigure>
        </PathGeometry.Figures>
      </PathGeometry>
    </GeometryGroup>
  </Path.Data>
</Path>

<Path Canvas.Left="0" Canvas.Top="0" Fill="Green" Stroke="Blue"
  StrokeThickness="2.5">
  <Path.Data>
    <GeometryGroup>
      <PathGeometry>
        <PathGeometry.Figures>
          <PathFigure StartPoint="160,70">
            <PathFigure.Segments>
              <BezierSegment Point1="175,85" Point2="200,99"
                Point3="215,100" />
              <LineSegment Point="195,148" />
              <BezierSegment Point1="174,150" Point2="142,140"
                Point3="129,115" />
              <LineSegment Point="160,70" />
            </PathFigure.Segments>
          </PathFigure>
        </PathGeometry.Figures>
      </PathGeometry>
    </GeometryGroup>
  </Path.Data>
</Path>
```

除了 BezierSegment 和 LineSegment 元素之外，还可以使用 ArcSegment 元素在两点之间绘制椭圆弧。使用 PolyLineSegment 可以绘制一组线段，PolyBezierSegment 由多条贝塞尔曲线组成，QuadraticBezierSegment 创建一条二次贝塞尔曲线，PolyQuadraticBezier- Segment 由多条二次贝塞尔曲线组成。

### 35.3.2　使用 PathMarkup 的几何图形

本章前面使用了路径标记和 Path 形状。在后台使用路径标记，可以通过 StreamGeometry 进行高效的绘图。UWP 应用程序的 XAML 会创建图形和片段。通过编程，可以创建线段、贝塞尔曲线和圆弧，以定义图形。通过 XAML 可以使用路径标记语法。路径标记语法可以与 Path 类的 Data 属性一起使用。特殊字符定义点的连接方式。在下面的示例中，M 标记起点，L 是到指定点的线条命令，Z 是闭合图形的闭合命令。图 35-4 显示了这个绘图操作的结果。路径标记语法允许使用更多的命令，如水平线(H)、垂直线(V)、三次贝塞尔曲线(C)、二次贝塞尔曲线(Q)、光滑的三次贝塞尔曲线(S)、光滑的二次贝塞尔曲线(T)，以及椭圆弧(A)(代码文件 Geometries/MainPage.xaml)：



图 35-4

```
<Path Canvas.Left="0" Canvas.Top="200" Fill="Yellow" Stroke="Blue"
  StrokeThickness="2.5"
  Data="M 120,5 L 128,80 L 220,50 L 160,130 L 190,220 L 100,150
    L 80,230 L 60,140 L0,110 L70,80 Z" StrokeLineJoin="Round">
</Path>
```

## 35.4　变换

因为 XAML 基于矢量，所以可以重置每个元素的大小。在下面的例子中，基于矢量的图形现在可以缩放、旋转和倾斜。不需要手工计算位置，就可以进行单击测试(如移动鼠标和鼠标单击)。

图 35-5 显示了一个矩形的几个不同形式。所有的矩形都定位在一个水平方向的 StackPanel 元素中，以并排放置矩形。第 1 个矩形有其原始大小和布局。第 2 个矩形重置了大小，第 3 个矩形移动了，第 4 个矩形旋转了，第 5 个矩形倾斜了，第 6 个矩形使用变换组进行变换，第 7 个矩形使用矩阵进行变换。下面各节讲述所有这些选项的代码示例。



图 35-5

### 35.4.1　缩放

给 Rectangle 元素的 RenderTransform 属性添加 ScaleTransform 元素，如下所示，把整个画布的内容在 X 轴方向上放大 0.5 倍，在 Y 轴方向上放大 0.4 倍(代码文件 Transformations/MainPage. xaml)。

```
<Rectangle Width="120" Height="60" Fill="Red" Margin="20">
  <Rectangle.RenderTransform>
    <ScaleTransform ScaleX="0.5" ScaleY="0.4" />
  </Rectangle.RenderTransform>
</Rectangle>
```

除了变换像矩形这样简单的形状之外，还可以变换任何 XAML 元素，因为 XAML 定义了矢量图形。在以下代码中，前面所示的脸部 Canvas 元素放在一个用户控件 SmilingFace 中，这个用户控件先显示没有转换的状

态，再显示调整大小后的状态。结果如图 35-6 所示。

```
<local:SmilingFace />
<local:SmilingFace>
  <local:SmilingFace.RenderTransform>
    <ScaleTransform ScaleX="1.6" ScaleY="0.8" CenterY="180" />
  </local:SmilingFace.RenderTransform>
</local:SmilingFace>
```



图 35-6

## 35.4.2 平移

在 X 轴或 Y 轴方向上移动一个元素时，可以使用 TranslateTransform。在以下代码片段中，给 X 指定-90，元素向左移动，给 Y 指定 20，元素向底部移动(代码文件 Transformations/MainPage.xaml)：

```
<Rectangle Width="120" Height="60" Fill="Green" Margin="20">
  <Rectangle.RenderTransform>
    <TranslateTransform X="-90" Y="20" />
  </Rectangle.RenderTransform>
</Rectangle>
```

## 35.4.3 旋转

使用 RotateTransform 元素，可以旋转元素。对于 RotateTransform，设置旋转的角度，用 CenterX 和 CenterY 设置旋转中心(代码文件 Transformation/MainPage.xaml )：

```
<Rectangle Width="120" Height="60" Fill="Orange" Margin="20">
  <Rectangle.RenderTransform>
    <RotateTransform Angle="45" CenterX="10" CenterY="-80" />
  </Rectangle.RenderTransform>
</Rectangle>
```

## 35.4.4 倾斜

对于倾斜，可以使用 SkewTransform 元素。此时可以指定 X 轴和 Y 轴方向的倾斜角度(代码文件 Transformation/MainPage.xaml )：

```
<Rectangle Width="120" Height="60" Fill="LightBlue" Margin="20">
  <Rectangle.RenderTransform>
    <SkewTransform AngleX="20" AngleY="30" CenterX="40" CenterY="390" />
  </Rectangle.RenderTransform>
</Rectangle>
```

## 35.4.5 组合变换和复合变换

同时执行多种变换的简单方式是使用 CompositeTransform 和 TransformationGroup 元素。TransformationGroup 元素可以包含 SkewTransform、RotateTransform、TranslateTransform 和 ScaleTransform 作为其子元素(代码文件 Transformations/MainPage. xaml)：

```
<Rectangle Width="120" Height="60" Fill="LightGreen" Margin="20">
  <Rectangle.RenderTransform>
    <TransformGroup>
      <SkewTransform AngleX="45" AngleY="20" CenterX="-390" CenterY="40" />
      <RotateTransform Angle="90" />
      <ScaleTransform ScaleX="0.5" ScaleY="1.2" />
    </TransformGroup>
  </Rectangle.RenderTransform>
</Rectangle>
```

为了同时执行旋转和倾斜操作，可以定义一个 TransformGroup，它同时包含 RotateTransform 和 SkewTransform。类 CompositeTransform 定义多个属性，用于一次进行多个变换。例如，ScaleX 和 ScaleY 进行缩放，TranslateX 和 TranslateY 移动元素。也可以定义一个 MatrixTransform，其中 Matrix 元素指定了用于拉伸的 M11 和 M22 属性，以及用于倾斜的 M12 和 M21 属性，见下一节。

### 35.4.6　使用矩阵的变换

同时执行多种变换的另一个选择是指定一个矩阵。这里使用 MatrixTransform.MatrixTransform 定义了 Matrix 属性，它有 6 个值。设置值 1,0,0,1,0,0 不改变元素。值 0.5,1.4,0.4,0.5,-200,0 会重置元素的大小、倾斜和平移元素(代码文件 Transformations/MainPage.xaml)：

```
<Rectangle Width="120" Height="60" Fill="Gold" Margin="20">
  <Rectangle.RenderTransform>
    <MatrixTransform Matrix="0.5, 1.4, 0.4, 0.5, -200, 0" />
  </Rectangle.RenderTransform>
</Rectangle>
```

如果将一个字符串赋给 Matrix 属性，则 MatrixTransform 类按顺序定义公共字段 M11、M12、M21、M22、OffsetX 和 OffsetY。MatrixTransform 实现了一个仿射变换，所以 9 个矩阵成员中只有 6 个需要指定。其余矩阵成员使用固定值 0、0、1。M11 和 M22 字段具有默认值 1，用于在 x 和 y 方向上伸缩。M12 和 M21 的默认值为 0，用于倾斜控件。OffsetX 和 OffsetY 的默认值为 0，用于平移控件。

## 35.5　画笔

本节演示了如何使用 XAML 的画笔绘制背景和前景。本节将学习如何使用纯色和线性渐变的笔刷，使用笔刷绘制图像，使用新的 AcrylicBrush 和 RevealBrush。

### 35.5.1　SolidColorBrush

图 35-7 中的第一个按钮使用了 SolidColorBrush，顾名思义，这支画笔使用纯色。全部区域用同一种颜色绘制。

把 Background 特性设置为定义纯色的字符串，就可以定义纯色。使用 BrushValueSerializer 把该字符串转换为一个 SolidColorBrush 元素(代码文件 Brushes/MainPage.xaml )。




图 35-7

```
<Button Background="#FFC9659C">Solid Color</Button>
```

当然，通过设置 Background 子元素并把 SolidColorBrush 元素添加为它的内容，也可以得到同样的效果。应用程序中的第一个按钮使用十六进制值用作纯背景色(代码文件 Brushes/MainPage.xaml)：

```
<Button Content="Solid Color 2">
  <Button.Background>
    <SolidColorBrush Color="#FFC9659C" />
  </Button.Background>
</Button>
```

### 35.5.2　LinearGradientBrush

对于平滑的颜色变化，可以使用 LinearGradientBrush，如图 35-8 所示。这个画笔定义了 StartPoint 和 EndPoint 属性。使用这些属性可以为线性渐变指定 2D 坐标。默认的渐变方向是从(0,0)到(1,1)的对角线。定义其他值可以给渐变指定不同的方向。例如，StartPoint 指定为(0,0)，EndPoint 指定为(0,1)，就得到了一个垂直渐变。StartPoint 和 EndPoint 值指定为(1,0)，就得到了一个水平渐变。

通过该画笔的内容，可以用 GradientStop 元素定义指定偏移位置的颜色值。在各个偏移位置之间，颜色是平滑过渡的(代码文件 Brushes/MainPage.xaml)。

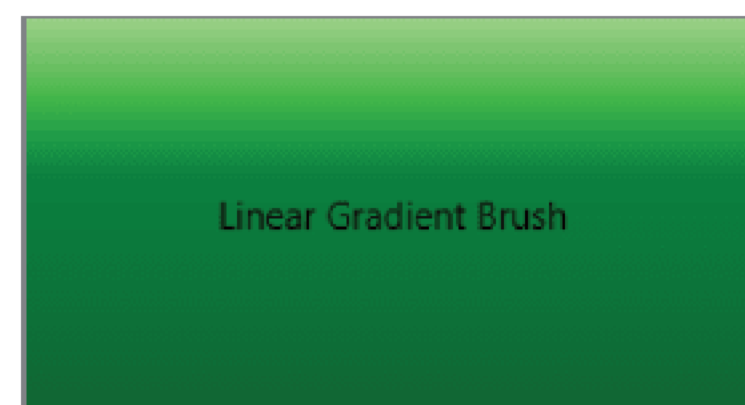



图 35-8

```
<Button Content="Linear Gradient Brush">
  <Button.Background>
    <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
```

```
        <GradientStop Offset="0" Color="LightGreen" />
        <GradientStop Offset="0.4" Color="Green" />
        <GradientStop Offset="1" Color="DarkGreen" />
      </LinearGradientBrush>
    </Button.Background>
  </Button>
```

## 35.5.3 ImageBrush

要把图像加载到画笔中，可以使用 ImageBrush 元素。通过这个元素，显示 ImageSource 属性定义的图像。图像可以在文件系统中访问，或从程序集的资源中访问。在代码示例中，添加文件系统中的图像(代码文件 Brushes/MainPage.xaml)：

```
  <Button Content="Image Brush" FontWeight="ExtraBold"
FontSize="28"
    RenderTransformOrigin="0.5,0.5" >
    <Button.Background>
      <ImageBrush ImageSource="msbuild.png" Opacity="0.5" />
    </Button.Background>
  </Button>
```

运行应用程序时，可以看到如图 35-9 所示的按钮。




图 35-9

## 35.5.4 AcrylicBrush

Windows 10 构建版 16299 的一个新画笔是 AcrylicBrush。这种画笔是新 Fluent Design 的一部分。AcrylicBrush 提供了透明效果，让应用或主机的其他元素通过该画笔显示出来。

发布为 Windows 10 一部分的一个应用程序是 Calculator。这个计算器有一些透明度，可以让其他应用程序或壁纸图像在应用程序中显示出来(参见图 35-10)。这种效果并不适用于计算器中的主数字按钮，但是计算器的其他元素可以让背景光线透出来。

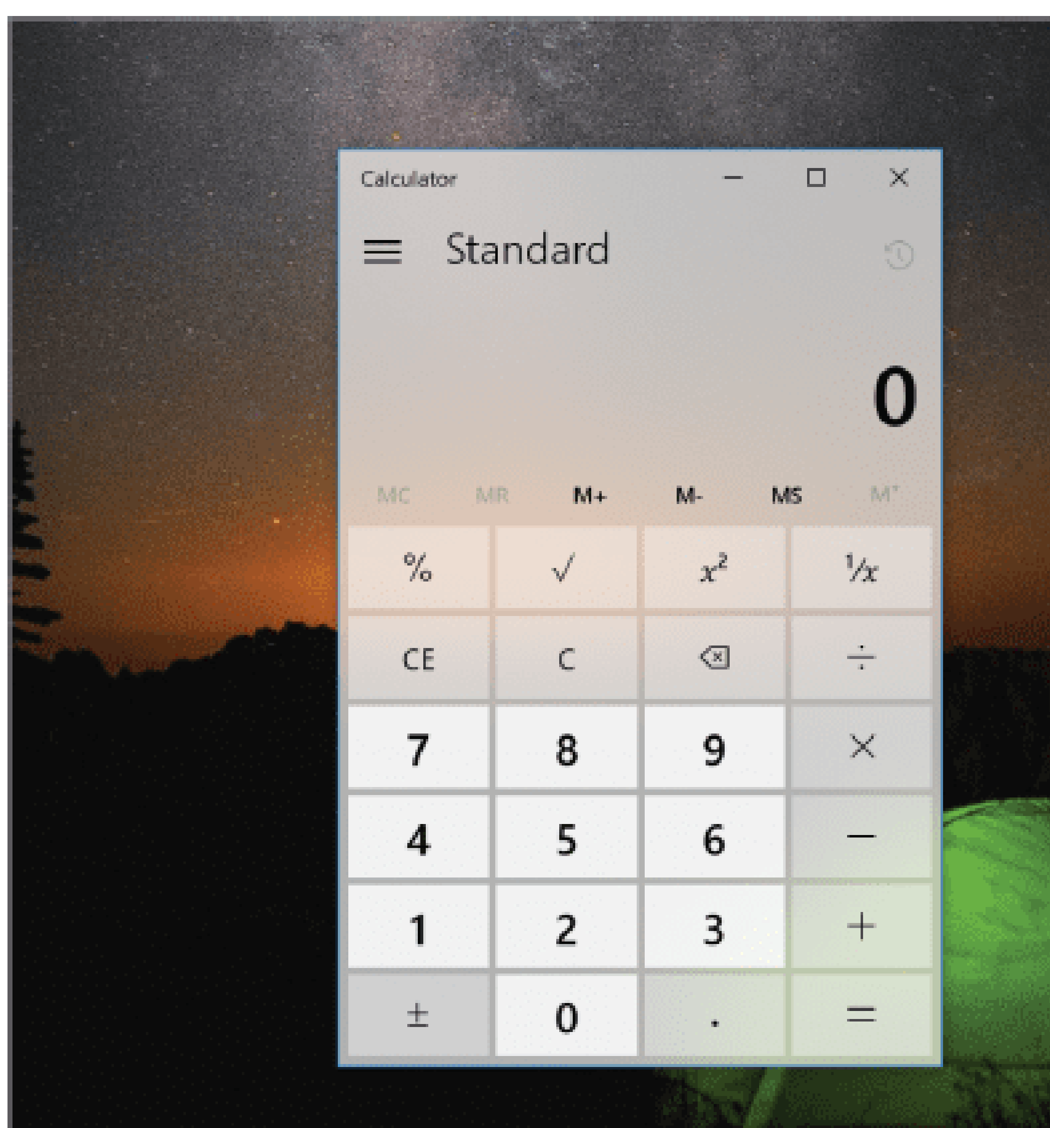



图 35-10

给按钮的 Background 属性分配了 AcrylicBrush。TintOpacity 的值取自滑块的值。这样，移动应用程序中的滑块时，就可以根据不透明度来查看画笔的不同效果。TintColor 属性指定笔刷的主色。

使用 BackgroundSource 属性，可以在 HostBackdrop 或 Backdrop 之间进行选择。使用 Backdrop 时，应用程序本身的颜色就会透出来。这就是所谓的 in-app acrylic。控件中使用该画笔覆盖的元素会显示出来。而使用 HostBackdrop 时，会选取应用程序下面的颜色，这就是 background acrylic。由于 acrylic 的 UI 效果需要 GPU 的功能，因此这一特性可以缩短电池寿命。当系统的功耗较低时，AcrylicBrush 使用由 FallbackColor 属性定义的纯色。还可以配置属性 AlwaysUseFallback 以始终使用 FallbackColor。用户配置可以触发此设置，以提高电池寿命(代码文件 Brushes/MainPage.xaml)：

```
  <Button Content="Acrylic Brush Host Backdrop">
```

```
<Button.Background>
  <AcrylicBrush BackgroundSource="HostBackdrop"
TintColor="#FFFF0000"
     TintOpacity="{x:Bind slider1.Value,
Mode=OneWay}"
     FallbackColor="Orange" />
</Button.Background>
</Button>
```

图 35-11 显示了当前的 TintOpacity 设置为 0.2 的 AcrylicBrush。顶部的按钮用 HostBackdrop 配置。这里可以看到应用程序下面的背景透出来。底部的按钮用 Backdrop 配置。这里可以看到按钮下面的一个橙色矩形透出来。

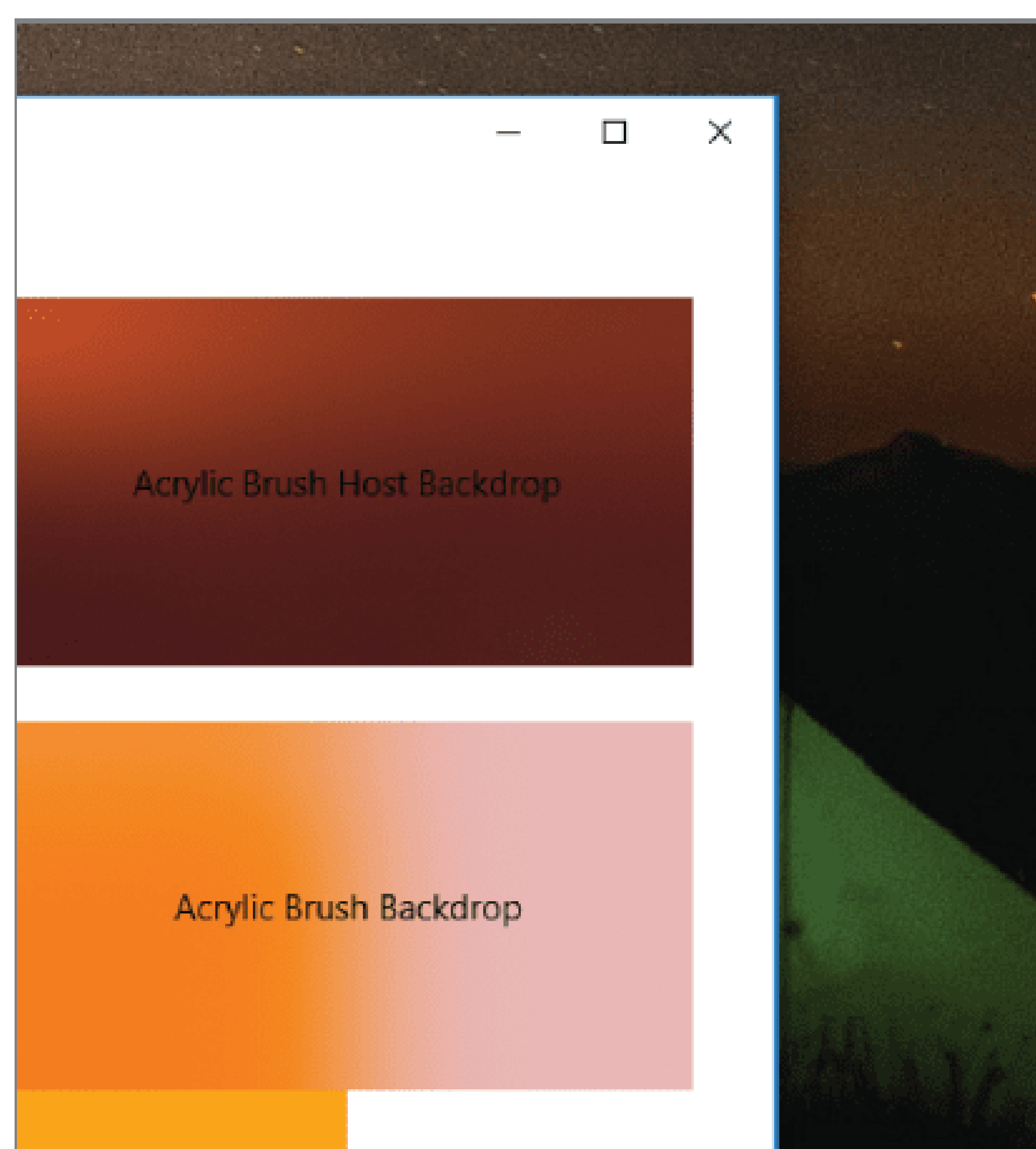


图 35-11

**注意：**
何时使用 acrylic 画笔？acrylic 给应用程序增加了纹理和深度。应用程序内的导航和命令在 acrylic 背景下看起来令人印象深刻。然而，应用程序的主要内容应该使用纯粹的背景。

### 35.5.5　RevealBrush

Windows 10 构建版 16299 的另一个新画笔是 RevealBrush。这个画笔突出显示用户使用灯光效果移动鼠标的区域。使用此画笔样式化按钮的一种简单方法是使用 ButtonRevealStyle，如下面的代码片段所示(代码文件 Brushes/MainPage.xaml)：

```
<Button Content="Button with Reveal Style" Padding="12" Margin="12"
  Style="{StaticResource ButtonRevealStyle}" />
```

**注意：**
样式和 StaticResource 标记扩展参见下一节。

使用 ButtonRevealStyle 时，需要仔细观察应用程序在运行时的效果。下面的按钮定义了一个较厚的边框，该边框使用主题资源 SystemColorControlHighlightAccentRevealBorderBrush 中定义的 BorderBrush，以帮助更清楚地看到效果。在按钮周围移动鼠标时，按钮的边框会突出显示出来。

```
<Button Margin="4" BorderThickness="8"
  Background="{ThemeResource SystemControlHighlightAccentRevealBackgroundBrush}"
  BorderBrush="{ThemeResource SystemControlHighlightAccentRevealBorderBrush}">
  With Reveal Border</Button>
```

## 35.6　样式和资源

设置 XAML 元素的 FontSize 和 Background 属性，就可以定义 XAML 元素的外观，如 Button 元素所示(代码文件 StylesAndResources/MainPage.xaml)：

```
<Button Width="150" FontSize="12" Background="AliceBlue" Content="Click Me!" />
```

除了定义每个元素的外观之外，还可以定义用资源存储的样式。为了完全定制控件的外观，可以使用模板，再把它们存储到资源中。

### 35.6.1　样式

控件的 Style 属性可以赋予附带 Setter 的 Style 元素。Setter 元素定义 Property 和 Value 属性，并给目标元素设置指定的属性和值。下例设置 Background、FontSize、FontWeight 和 Margin 属性。把 Style 设置为 TargetType

Button，以便直接访问 Button 的属性(代码文件 StylesAndResources/MainPage.xaml)。

```
<Button Width="150" Content="Click Me!">
  <Button.Style>
    <Style TargetType="Button">
      <Setter Property="Background" Value="Yellow" />
      <Setter Property="FontSize" Value="14" />
      <Setter Property="FontWeight" Value="Bold" />
      <Setter Property="Margin" Value="5" />
    </Style>
  </Button.Style>
</Button>
```

直接通过 Button 元素设置 Style 对样式的共享没有什么帮助。样式可以放在资源中。在资源中，可以把样式赋予指定的元素，把一个样式赋予某一类型的所有元素，或者为该样式使用一个键。要把样式赋予某一类型的所有元素，可使用 Style 的 TargetType 属性，将样式赋予一个按钮。要定义需要引用的样式，必须设置 x:Key：

```
<Page.Resources>
  <Style TargetType="Button">
    <Setter Property="Background" Value="LemonChiffon" />
    <Setter Property="FontSize" Value="18" />
    <Setter Property="Margin" Value="5" />
  </Style>
  <Style x:Key="ButtonStyle1" TargetType="Button">
    <Setter Property="Background" Value="Red" />
    <Setter Property="Foreground" Value="White" />
    <Setter Property="FontSize" Value="18" />
    <Setter Property="Margin" Value="5" />
  </Style>
</Page.Resources>
```

在样例应用程序中，在页面内全局定义的样式在 Page 元素的 Resources 属性中指定。

在下面的 XAML 代码中，第一个按钮没有用元素属性定义样式，而是使用为 Button 类型定义的样式。对于下一个按钮，把 Style 属性用 StaticResource 标记扩展设置为{StaticResource ButtonStyle}，而 ButtonStyle 指定了前面定义的样式资源的键值，所以该按钮的背景为红色，前景是白色。

```
<Button Width="200" Content="Default Button style" Margin="3" />
<Button Width="200" Content="Named style"
  Style="{StaticResource ButtonStyle1}" Margin="3" />
```

除了把按钮的 Background 设置为单个值之外，还可以将 Background 属性设置为定义了渐变色的 LinearGradientBrush，如下所示：

```
<Style x:Key="FancyButtonStyle" TargetType="Button">
  <Setter Property="FontSize" Value="22" />
  <Setter Property="Foreground" Value="White" />
  <Setter Property="Margin" Value="5" />
  <Setter Property="Background">
    <Setter.Value>
      <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
        <GradientStop Offset="0.0" Color="LightCyan" />
        <GradientStop Offset="0.14" Color="Cyan" />
        <GradientStop Offset="0.7" Color="DarkCyan" />
      </LinearGradientBrush>
    </Setter.Value>
  </Setter>
</Style>
```

本例中下一个按钮的样式采用青色的线性渐变效果：

```
<Button Width="200" Content="Fancy button style"
  Style="{StaticResource FancyButtonStyle}" Margin="3" />
```

样式提供了一种继承方式。一个样式可以基于另一个样式。下面的 AnotherButtonStyle 样式基于 FancyButtonStyle 样式。它使用该样式定义的所有设置，且通过 BasedOn 属性引用，但 Foreground 属性除外，它设置为 LinearGradientBrush：

```
<Style x:Key="AnotherButtonStyle" BasedOn="{StaticResource FancyButtonStyle}"
  TargetType="Button">
  <Setter Property="Foreground">
    <Setter.Value>
      <LinearGradientBrush>
```

```
        <GradientStop Offset="0.2" Color="White" />
        <GradientStop Offset="0.5" Color="LightYellow" />
        <GradientStop Offset="0.9" Color="Orange" />
      </LinearGradientBrush>
    </Setter.Value>
  </Setter>
</Style>
```

最后一个按钮应用了 AnotherButtonStyle：

```
<Button Width="200" Content="Style inheritance"
  Style="{StaticResource AnotherButtonStyle}" Margin="3" />
```

图 35-12 显示了所有这些按钮样式化后的效果。

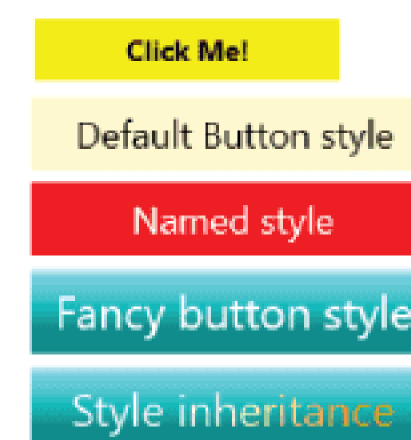



图 35-12

## 35.6.2　资源

从样式示例可以看出，样式通常存储在资源中。可以在资源中定义任意可共享的元素。例如，前面为按钮的背景样式创建了画笔，它本身就可以定义为一个资源，这样就可以在需要画笔的地方使用它。

下面的示例在 StackPanel 资源中定义一个 LinearGradientBrush，它的键名是 MyGradientBrush。button1 使用 StaticResource 标记扩展将 Background 属性赋予 MyGradientBrush 资源(代码文件 StylesAndResources/Resource DemoPage .xaml)：

```
<StackPanel x:Name="myContainer">
  <StackPanel.Resources>
    <LinearGradientBrush x:Key="MyGradientBrush" StartPoint="0,0"
      EndPoint="0.3,1">
      <GradientStop Offset="0.0" Color="LightCyan" />
      <GradientStop Offset="0.14" Color="Cyan" />
      <GradientStop Offset="0.7" Color="DarkCyan" />
    </LinearGradientBrush>
  </StackPanel.Resources>
  <Button Width="200" Height="50" Foreground="White" Margin="5"
    Background="{StaticResource MyGradientBrush}" Content="Click Me!" />
</StackPanel>
```

这里，资源用 StackPanel 定义。在上面的例子中，资源用 Page 或 Window 元素定义。基类 FrameworkElement 定义 ResourceDictionary 类型的 Resources 属性。这就是资源可以用派生自 FrameworkElement 的所有类(任意 XAML 元素)来定义的原因。

资源按层次结构来搜索。如果用根元素定义资源，它就会应用于所有子元素。如果根元素包含一个 Grid，该 Grid 包含一个 StackPanel，且资源是用 StackPanel 定义的，该资源就会应用于 StackPanel 中的所有控件。如果 StackPanel 包含一个按钮，但只用该按钮定义资源，这个样式就只对该按钮有效。

**注意：**

*对于层次结构，需要注意是否为样式使用了没有 Key 的 TargetType。如果用 Canvas 元素定义一个资源，并把样式的 TargetType 设置为应用于 TextBox 元素，该样式就会应用于 Canvas 中的所有 TextBox 元素。如果 Canvas 中有一个 ListBox，该样式甚至会应用于 ListBox 包含的 TextBox 元素。*

如果需要将同一个样式应用于多个窗口，就可以用应用程序定义样式。在用 Visual Studio 创建的 Windows 应用程序中，创建 App.xaml 文件，以定义应用程序的全局资源。应用程序样式对其中的每个页面或窗口都有效。每个元素都可以访问用应用程序定义的资源。如果通过父窗口找不到资源，就可以通过 Application 继续搜索资源(代码文件 StylesAndResourcesUWP/App.xaml)：

```
<Application x:Class="StylesAndResources.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  RequestedTheme="Light">
  <Application.Resources>
  </Application.Resources>
</Application>
```

### 35.6.3 从代码中访问资源

要从代码隐藏中访问资源，基类 FrameworkElement 的 Resources 属性返回 ResourceDictionary。该字典使用索引器和资源名称提供对资源的访问。可以使用 ContainsKey 方法检查资源是否可用。

下面看一个例子。按钮控件 button1 没有指定背景，但将 Click 事件动态赋予 OnApplyResource()方法，以动态修改它(代码文件 StylesAndResources/ResourceDemoPage.xaml)：

```
<Button Name="button1" Width="220" Height="50" Margin="5"
  Click="OnApplyResources" Content="Apply Resource Programmatically" />
```

使用 WPF 时，使用 TryFindResource 方法来迭代所有资源。使用 UWP 时，可以使用类似的方法，但是需要自己实现它。OnApplyResources 方法调用扩展方法 TryFindResource 来查找名为 MyGradientBrush 的资源，并将其分配给控件的 Background 属性(代码文件 StylesAndResources/ResourceDemo.xaml.cs)：

```
private void OnApplyResources(object sender, RoutedEventArgs e)
{
  if (sender is Control ctrl)
  {
    ctrl.Background = ctrl.TryFindResource("MyGradientBrush") as Brush;
  }
}
```

方法 TryFindResource 使用 ContainsKey 检查请求的资源是否可用，它会递归地调用方法，以免资源还没有找到(代码文件 StylesAndResources/FrameworkElementExtensions.cs)：

```
public static class FrameworkElementExtensions
{
  public static object TryFindResource(this FrameworkElement e, string key)
  {
    if (e == null) throw new ArgumentNullException(nameof(e));
    if (key == null) throw new ArgumentNullException(nameof(key));

    if (e.Resources.ContainsKey(key))
    {
      return e.Resources[key];
    }
    else if (e.Parent is FrameworkElement parent)
    {
      return TryFindResource(parent, key);
    }
    else
    {
      return null;
    }
  }
}
```

### 35.6.4 资源字典

如果相同的资源可用于不同的页面甚至不同的应用程序，把资源放在一个资源字典中就比较有效。使用资源字典，可以在多个应用程序之间共享文件，也可以把资源字典放在一个程序集中，供应用程序共享。

要共享程序集中的资源字典，应创建一个库。可以把资源字典文件(这里是 Dictionary1.xaml)添加到程序集中。

Dictionary1.xaml 定义了两个资源：一个是包含 CyanGradientBrush 键的 LinearGradientBrush，另一个是用于按钮的样式，它可以通过 PinkButtonStyle 键来引用(代码文件 ResourcesLib/Dictionary1.xaml)：

```
<ResourceDictionary
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <LinearGradientBrush x:Key="CyanGradientBrush" StartPoint="0,0"
    EndPoint="0.3,1">
    <LinearGradientBrush.GradientStops>
      <GradientStop Offset="0.0" Color="LightCyan" />
      <GradientStop Offset="0.14" Color="Cyan" />
      <GradientStop Offset="0.7" Color="DarkCyan" />
    </LinearGradientBrush.GradientStops>
```

```
    </LinearGradientBrush>

    <Style x:Key="PinkButtonStyle" TargetType="Button">
      <Setter Property="FontSize" Value="22" />
      <Setter Property="Foreground" Value="White" />
      <Setter Property="Background">
        <Setter.Value>
          <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
            <LinearGradientBrush.GradientStops>
              <GradientStop Offset="0.0" Color="Pink" />
              <GradientStop Offset="0.3" Color="DeepPink" />
              <GradientStop Offset="0.9" Color="DarkOrchid" />
            </LinearGradientBrush.GradientStops>
          </LinearGradientBrush>
        </Setter.Value>
      </Setter>
    </Style>
  </ResourceDictionary>
```

对于目标项目，需要引用这个库，并把资源字典添加到这个字典中。通过 ResourceDictionary 的 MergedDictionaries 属性，可以使用添加进来的多个资源字典文件。可以把一个资源字典列表添加到合并的字典中。对于 UWP 应用程序，引用的资源字典必须以 ms-appx:///模式作为前缀(代码文件 StylesAndResources/App.xaml)：

```
<Application x:Class="StylesAndResources.App"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:local="using:StylesAndResources"
RequestedTheme="Light">
  <Application.Resources>
    <ResourceDictionary>
      <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary
          Source="ms-appx:///ResourcesLibUWP/Dictionary1.xaml" />
      </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
  </Application.Resources>
</Application>
```

现在可以像本地资源那样使用引用程序集中的资源了(代码文件 StylesAndResources/ResourceDemoPage.xaml)：

```
<Button Width="300" Height="50" Style="{StaticResource PinkButtonStyle}"
  Content="Referenced Resource" />
```

### 35.6.5　主题资源

WPF 支持 DynamicResource 标记扩展，在应用程序运行时，如果资源发生变化，它会动态更新用户界面。尽管 UWP 应用程序不支持 DynamicResource 标记扩展，但这些应用程序也能动态改变样式。这个功能是基于主题的。通过主题，可以允许用户在光明与黑暗主题之间切换(类似于可以用 Visual Studio 改变的主题)。

#### 1. 定义主题资源

主题资源可以在 ThemeDictionaries 集合的资源字典中定义。在 ThemeDictionaries 集合中定义的 ResourceDictionary 对象需要分配一个包含主题名称(Light 或 Dark)的键。示例代码为浅色背景和暗色前景的 Light 主题定义了一个按钮，为浅色前景和暗色背景的 dark 主题定义了一个按钮。用于样式的键在这两个字典中是一样的：SampleButtonStyle(代码文件 StylesAndResources/Styles/SampleThemes.xaml)：

```
<ResourceDictionary
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:StylesAndResourcesUWP">
  <ResourceDictionary.ThemeDictionaries>
    <ResourceDictionary x:Key="Light">
      <Style TargetType="Button" x:Key="SampleButtonStyle">
        <Setter Property="Background" Value="LightGray" />
        <Setter Property="Foreground" Value="Black" />
      </Style>
    </ResourceDictionary>
```

```
    <ResourceDictionary x:Key="Dark">
      <Style TargetType="Button" x:Key="SampleButtonStyle">
        <Setter Property="Background" Value="Black" />
        <Setter Property="Foreground" Value="White" />
      </Style>
    </ResourceDictionary>
  </ResourceDictionary.ThemeDictionaries>
</ResourceDictionary>
```

使用 ThemeResource 标记扩展可以指定样式。除了使用另一个标记扩展之外，其他的都与 StaticResource 标记扩展相同(代码文件 StylesAndResourcesUWP/ThemeDemoPage.xaml)：

```
<Button Style="{ThemeResource SampleButtonStyle}" Click="OnChangeTheme"
  Content="Change Theme" />
```

根据选择的主题，使用相应的样式。

### 2. 选择主题

有不同的方式选择主题。首先，应用程序本身有一个默认的主题。Application类的 RequestedTheme属性定义了应用程序的默认主题。这在App.xaml内定义，在其中还引用了主题字典文件(代码文件StylesAndResources/App.xaml)：

```
<Application
  x:Class="StylesAndResourcesUWP.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:StylesAndResourcesUWP"
  RequestedTheme="Light">
  <Application.Resources>
    <ResourceDictionary>
      <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary Source="ms-appx:///StylesLib/Dictionary1.xaml" />
        <ResourceDictionary Source="Styles/SampleThemes.xaml" />
      </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
  </Application.Resources>
</Application>
```

RequestedTheme 属性在 XAML 元素的层次结构中定义。每个元素可以覆盖用于它本身及其子元素的主题。下面的 Grid 元素改变了 Dark 主题的默认主题。现在它用于 Grid 元素及其所有子元素(代码文件 StylesAndResources/ThemeDemoPage.xaml)：

```
<Grid x:Name="grid1"
  Background="{ThemeResource ApplicationPageBackgroundThemeBrush}"
  RequestedTheme="Dark">
  <Button Style="{ThemeResource SampleButtonStyle}" Click="OnChangeTheme"
    Content="Change Theme" />
</Grid>
```

也可以在代码中通过设置 RequestedTheme 属性来动态更改主题(代码文件 StylesAndResources/ThemeDemoPage.xaml.cs)：

```
private void OnChangeTheme(object sender, RoutedEventArgs e)
{
  grid1.RequestedTheme = grid1.RequestedTheme == ElementTheme.Dark ?
  ElementTheme.Light: ElementTheme.Dark;
}
```

**注意：**
只有资源看起来与主题不同，使用 ThemeResource 标记扩展才有用。如果资源应该与主题相同，就应继续使用 StaticResource 标记扩展。

## 35.7 模板

XAML Button 控件可以包含任何内容，如简单的文本，还可以给按钮添加 Canvas 元素，Canvas 元素可以

包含形状，也可以给按钮添加 Grid 或视频。然而，按钮还可以完成更多的操作。使用基于模板的 XAML 控件，控件的外观及其功能在 WPF 中是完全分离的。虽然按钮有默认的外观，但可以用模板完全定制其外观。

如表 35-2 所示，Windows 应用程序提供了几个模板类型，它们派生自基类 FrameworkTemplate。

表 35-2

| 模 板 类 型 | 说　明 |
|---|---|
| ControlTemplate | 使用 ControlTemplate 可以指定控件的可视化结构，重新设计其外观 |
| ItemsPanelTemplate | 对于 ItemsControl，可以赋予一个 ItemsPanelTemplate，以指定其项的布局。每个 ItemsControl 都有一个默认的 ItemsPanelTemplate。MenuItem 使用 WrapPanel，StatusBar 使用 DockPanel，ListBox 使用 VirtualizingStackPanel |
| DataTemplate | DataTemplate非常适用于对象的图形表示。给列表框指定样式时，默认情况下，列表框中的项根据 ToString()方法的输出来显示。应用DataTemplate，可以重写其操作，定义项的自定义表示 |

## 35.7.1　控件模板

本章前面介绍了如何给控件的属性定义样式。如果设置控件的简单属性得不到需要的外观，就可以修改 Template 属性。使用 Template 属性可以定制控件的整体外观。下面的例子说明了定制按钮的过程，后面逐步地说明了列表框的定制，以便显示出改变的中间结果。

Button 类型的定制在一个单独的资源字典文件 ControlTemplates.xaml 中进行。这里定义了键名为 RoundedGelButton 的样式。RoundedGelButton 样式设置 Background、Height、Foreground、Margin 和 Template 属性。Template 属性是这个样式中最有趣的部分，它指定一个仅包含一行一列的网格。

在这个单元格中，有一个名为 GelBackground 的椭圆。这个椭圆给笔触设置了一个线性渐变画笔。包围矩形的笔触非常细，因为把 StrokeThickness 设置为 0.5。

因为第二个椭圆 GelShine 比较小，其尺寸由 Margin 属性定义，所以在第一个椭圆内部是可见的。因为其笔触是透明的，所以该椭圆没有边框。这个椭圆使用一个线性渐变填充画笔，从部分透明的浅色变为完全透明，这使椭圆具有“亦真亦幻”的效果(代码文件 Templates/Styles/ControlTemplates.xaml)：

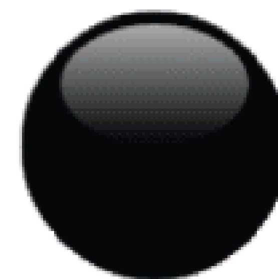
```
<ResourceDictionary
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

  <Style x:Key="RoundedGelButton" TargetType="Button">
    <Setter Property="Width" Value="100" />
    <Setter Property="Height" Value="100" />
    <Setter Property="Foreground" Value="White" />
    <Setter Property="Template">
      <Setter.Value>
        <ControlTemplate TargetType="Button">
          <Grid>
            <Ellipse Name="GelBackground" StrokeThickness="0.5" Fill="Black">
              <Ellipse.Stroke>
                <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                  <GradientStop Offset="0" Color="#ff7e7e7e" />
                  <GradientStop Offset="1" Color="Black" />
                </LinearGradientBrush>
              </Ellipse.Stroke>
            </Ellipse>
            <Ellipse Margin="15,5,15,50">
              <Ellipse.Fill>
                <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                  <GradientStop Offset="0" Color="#aaffffff" />
                  <GradientStop Offset="1" Color="Transparent" />
                </LinearGradientBrush>
              </Ellipse.Fill>
            </Ellipse>
          </Grid>
        </ControlTemplate>
      </Setter.Value>
```

```
      </Setter>
    </Style>
</ResourceDictionary>
```

从 app.xaml 文件中，引用资源字典，如下所示(代码文件 Template/App.xaml)：

```
<Application x:Class="TemplateDemo.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  StartupUri="MainWindow.xaml">
  <Application.Resources>
    <ResourceDictionary Source="Styles/ControlTemplates.xaml" />
  </Application.Resources
</Application>
```


图 35-13

现在可以把 Button 控件关联到样式上。按钮的新外观如图 35-13 所示，并使用代码文件 Templates /StyledButtons.xaml。

```
<Button Style="{StaticResource RoundedGelButton}" Content="Click Me!" />
```



图 35-14

按钮现在的外观完全不同，但按钮的内容未在图 35-13 中显示出来。必须扩展前面创建的模板，以把按钮的内容显示在新外观上。为此需要添加一个 ContentPresenter。ContentPresenter 是控件内容的占位符，并定义了放置这些内容的位置。这里把内容放在网格的第一行上，即 Ellipse 元素所在的位置。ContentPresenter 的 Content 属性定义了内容的外观。把内容设置为 TemplateBinding 标记表达式。TemplateBinding 绑定父模板，这里是 Button 元素。{TemplateBinding Content}指定，Button 控件的 Content 属性值应作为内容放在占位符内。图 35-14 显示了带内容的按钮(代码文件 Templates/Styles/ControlTemplates.xaml)：

```
<Setter Property="Template">
  <Setter.Value>
    <ControlTemplate TargetType="Button">
      <Grid>
        <Ellipse Name="GelBackground" StrokeThickness="0.5" Fill="Black">
          <Ellipse.Stroke>
            <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
              <GradientStop Offset="0" Color="#ff7e7e7e" />
              <GradientStop Offset="1" Color="Black" />
            </LinearGradientBrush>
          </Ellipse.Stroke>
        </Ellipse>
        <Ellipse Margin="15,5,15,50">
          <Ellipse.Fill>
            <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
              <GradientStop Offset="0" Color="#aaffffff" />
              <GradientStop Offset="1" Color="Transparent" />
            </LinearGradientBrush>
          </Ellipse.Fill>
        </Ellipse>
        <ContentPresenter Name="GelButtonContent"
          VerticalAlignment="Center"
          HorizontalAlignment="Center"
          Content="{TemplateBinding Content}" />
      </Grid>
    </ControlTemplate>
  </Setter.Value>
```

**注意：**
TemplateBinding 允许与模板交流控件定义的值。这不仅可以用于内容，还可以用于颜色和笔触样式等。

现在这样一个样式化的按钮在屏幕上看起来很漂亮。但仍有一个问题：如果用鼠标单击该按钮，或使鼠标滑过该按钮，则它不会有任何动作。这不是用户操作按钮时的一般情况。解决方法如下：对于模板样式的按钮，必须给它指定可视化状态或触发器，使按钮在响应鼠标移动和鼠标单击时有不同的外观。可视化状态也利用动画，因此本章后面讨论这个变更。

然而，为了提前了解这一点，可以使用 Visual Studio 创建一个按钮模板。不是完全从头开始建立这样一个模板，而可以在 XAML 设计器或文档浏览器中选择一个按钮控件，从上下文菜单中选择 Edit Template。在这里，可以创建一个空的模板，或复制预定义的模板。使用模板的一个副本来查看预定义的模板。创建一个样式资源

的对话框参见图 35-15。在这里可以定义包含模板的资源是在文档、应用程序(用于多个页面和窗口)还是资源字典中创建。对于之前样式化的按钮，资源字典 ControlTemplates.xaml 已经存在，示例代码在该字典中创建资源。

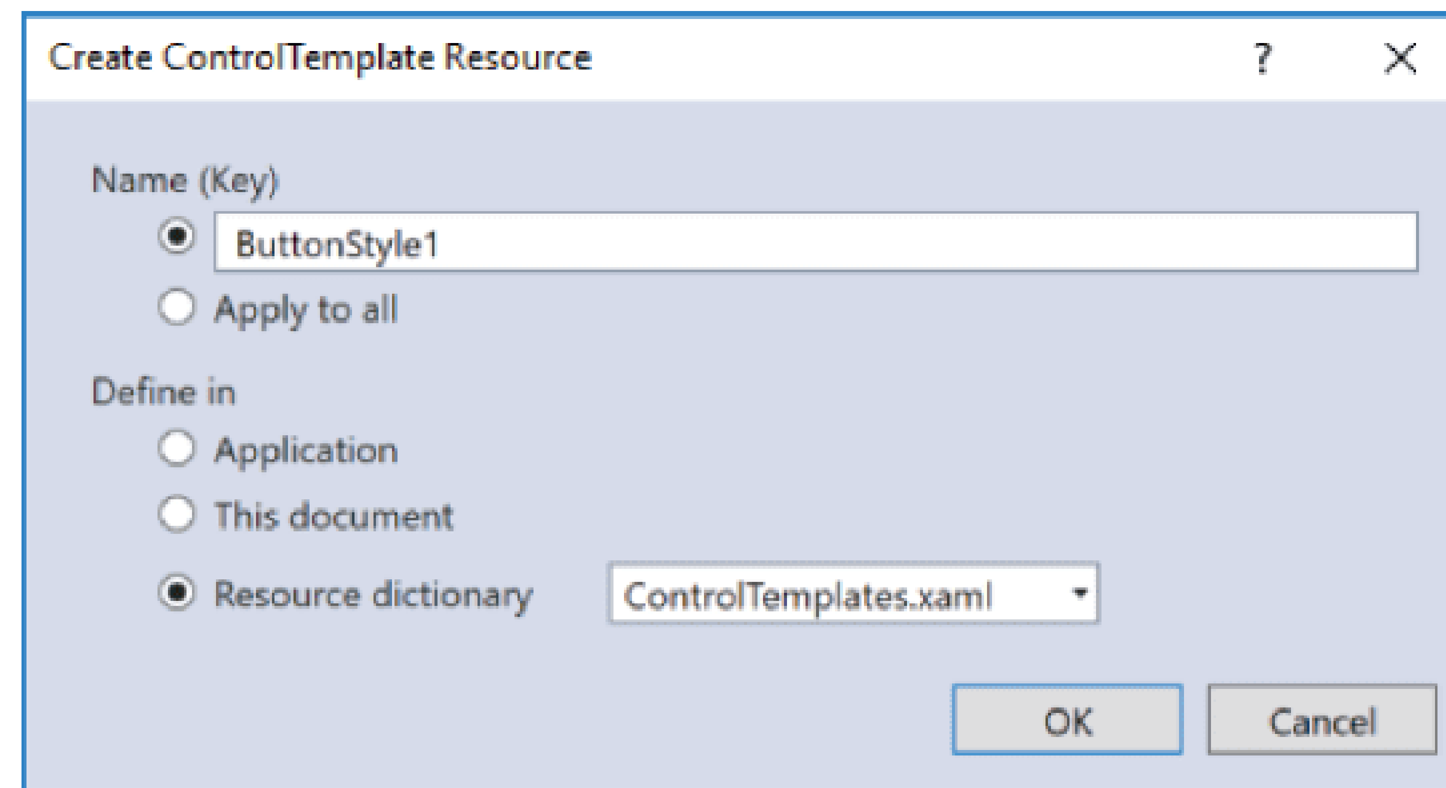



图 35-15

以下代码片段显示了 Windows 应用程序中默认按钮模板的一些特殊之处。几个按钮设置取自主题资源，如 Background、Foreground 和 BorderBrush。它们在光明和黑暗主题中是不同的。一些值，如 Padding 和 HorizontalAlignment 是固定的。创建一个自定义样式，就可以改变这些(代码文件 Templates/Styles/ControlTemplates.xaml)：

```
<Style x:Key="ButtonStyle1" TargetType="Button">
  <Setter Property="Background" Value="{ThemeResource ButtonBackground}"/>
  <Setter Property="Foreground" Value="{ThemeResource ButtonForeground}"/>
  <Setter Property="BorderBrush" Value="{ThemeResource ButtonBorderBrush}"/>
  <Setter Property="BorderThickness"
    Value="{ThemeResource ButtonBorderThemeThickness}"/>
  <Setter Property="Padding" Value="8,4,8,4"/>
  <Setter Property="HorizontalAlignment" Value="Left"/>
  <Setter Property="VerticalAlignment" Value="Center"/>
  <Setter Property="FontFamily"
    Value="{ThemeResource ContentControlThemeFontFamily}"/>
  <Setter Property="FontWeight" Value="Normal"/>
  <Setter Property="FontSize"
    Value="{ThemeResource ControlContentThemeFontSize}"/>
  <Setter Property="UseSystemFocusVisuals" Value="True"/>
  <Setter Property="FocusVisualMargin" Value="-3"/>
```

控件模板由一个 Grid 网格和一个 ContentPresenter 组成，画笔和边界值使用 TemplateBinding 限定。这样就可以用按钮控件直接定义这些值，来影响外观。

```
<Setter Property="Template">
  <Setter.Value>
    <ControlTemplate TargetType="Button">
      <Grid x:Name="RootGrid" Background="{TemplateBinding Background}">
        <!-Visual State Manager settings removed->
        <ContentPresenter x:Name="ContentPresenter"
          AutomationProperties.AccessibilityView="Raw"
          BorderBrush="{TemplateBinding BorderBrush}"
          BorderThickness="{TemplateBinding BorderThickness}"
          ContentTemplate="{TemplateBinding ContentTemplate}"
          ContentTransitions="{TemplateBinding ContentTransitions}"
          Content="{TemplateBinding Content}"
          HorizontalContentAlignment=
            "{TemplateBinding HorizontalContentAlignment}"
          Padding="{TemplateBinding Padding}"
          VerticalContentAlignment=
            "{TemplateBinding VerticalContentAlignment}"/>
      </Grid>
    </ControlTemplate>
  </Setter.Value>
</Setter>
```

对于动态更改按钮，如果鼠标划过按钮，或按钮被按下，应用程序的按钮模板就会利用 VisualStateManager。在这里，按钮的状态改为 PointerOver、Pressed 和 Disabled 时，就定义关键帧动画。

```
<VisualStateManager.VisualStateGroups>
  <VisualStateGroup x:Name="CommonStates">
```

```
        <VisualState x:Name="Normal">
          <Storyboard>
            <PointerUpThemeAnimation Storyboard.TargetName="RootGrid"/>
          </Storyboard>
        </VisualState>
        <VisualState x:Name="PointerOver">
          <Storyboard>
            <ObjectAnimationUsingKeyFrames
              Storyboard.TargetProperty="BorderBrush"
              Storyboard.TargetName="ContentPresenter">
              <DiscreteObjectKeyFrame KeyTime="0"
                Value="{ThemeResource SystemControlHighlightBaseMediumLowBrush}"/>
            </ObjectAnimationUsingKeyFrames>
            <ObjectAnimationUsingKeyFrames
              Storyboard.TargetProperty="Foreground"
              Storyboard.TargetName="ContentPresenter">
              <DiscreteObjectKeyFrame KeyTime="0"
                Value="{ThemeResource SystemControlHighlightBaseHighBrush}"/>
            </ObjectAnimationUsingKeyFrames>
          </Storyboard>
        </VisualState>
        <VisualState x:Name="Pressed">
          <Storyboard>
            <!-animations removed->
          </Storyboard>
        </VisualState>
        <VisualState x:Name="Disabled">
          <Storyboard>
            <!-animations removed->
          </Storyboard>
        </VisualState>
      </VisualStateGroup>
    </VisualStateManager.VisualStateGroups>
```

## 35.7.2 数据模板

ContentControl 元素的内容可以是任意内容——不仅可以是 XAML 元素，还可以是.NET 对象。例如，可以把 Country 类型的对象赋予 Button 类的内容。下面的示例创建 Country 类，以表示国家名称和国旗(用一幅图像的路径表示)。这个类定义 Name 和 ImagePath 属性，并重写 ToString()方法，用于默认的字符串表示(代码文件 Models/Country.cs)：

```
public class Country
{
  public string Name { get; set; }
  public string ImagePath { get; set; }
  public override string ToString() => Name;
}
```

这些内容在按钮或任何其他 ContentControl 中会如何显示？默认情况下会调用 ToString()方法，显示对象的字符串表示。

要获得自定义外观，还可以为 Country 类型创建一个 DataTemplate。示例代码定义了 CountryDataTemplate 键，这个键可以用于引用模板。在 DataTemplate 内部，主元素是一个文本框，其 Text 属性绑定到 Country 的 Name 属性上，Source 属性的 Image 绑定到 Country 的 ImagePath 属性上。Grid 和 Border 元素定义了布局和可见外观(代码文件 Templates/Styles/DataTemplates.xaml)：

```
<DataTemplate x:Key="CountryDataTemplate">
  <Border Margin="4" BorderThickness="2" CornerRadius="6">
    <Border.BorderBrush>
      <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
        <GradientStop Offset="0" Color="#aaa" />
        <GradientStop Offset="1" Color="#222" />
      </LinearGradientBrush>
    </Border.BorderBrush>
  <Border.Background>
    <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
      <GradientStop Offset="0" Color="#444" />
      <GradientStop Offset="1" Color="#fff" />
    </LinearGradientBrush>
  </Border.Background>
    <Grid Margin="4">
```

```
    <Grid.RowDefinitions>
      <RowDefinition Height="auto" />
      <RowDefinition Height="auto" />
    </Grid.RowDefinitions>
    <Image Width="120" Source="{Binding ImagePath}" />
    <TextBlock Grid.Row="1" Opacity="0.6" FontSize="16"
      VerticalAlignment="Bottom" HorizontalAlignment="Right" Margin="15"
      FontWeight="Bold" Text="{Binding Name}" />
    </Grid>
  </Border>
</DataTemplate>
```

在 Page 的 XAML 代码中，定义一个简单的 Button 元素 button1：

```
<Button x:Name="countryButton" Grid.Row="2" Margin="20"
  ContentTemplate="{StaticResource CountryDataTemplate}" />
```

在代码隐藏文件中，实例化一个新的 Country 对象，并把它赋给 button1 的 Content 属性(代码文件 Templates/StyledButtons.xaml.cs)：

```
this.countryButton.Content = new Country
{
  Name = "Austria",
  ImagePath = "images/Austria.bmp"
};
```

运行这个应用程序，可以看出，DataTemplate 应用于 Button，因为 Country 数据类型有默认的模板，如图 35-16 所示。

当然，也可以创建一个控件模板，并从中使用数据模板。




图 35-16

## 35.7.3　样式化 ListView

更改按钮或标签的样式是一个简单的任务，例如改变包含一个元素列表的父元素的样式。如何更改 ListView？这个列表控件也有操作方式和外观。它可以显示一个元素列表，用户可以从列表中选择一个或多个元素。至于操作方式，ListView 类定义了方法、属性和事件。ListView 的外观与其操作是分开的。ListView 元素有一个默认的外观，但可以通过创建模板，改变这个外观。

为了给 ListView 填充一些项，类 CountryRepository 返回几个要显示出来的国家(代码文件 Models/CountryRepository.cs)：

```
public sealed class CountryRepository
{
  private static IEnumerable<Country> s_countries;
  public IEnumerable<Country> GetCountries() =>
    s_countries ?? (s_countries = new List<Country>
    {
      new Country { Name="Austria", ImagePath = "Images/Austria.bmp" },
      new Country { Name="Germany", ImagePath = "Images/Germany.bmp" },
      new Country { Name="Norway", ImagePath = "Images/Norway.bmp" },
      new Country { Name="USA", ImagePath = "Images/USA.bmp" }
    });
}
```

在代码隐藏文件中，在 StyledList 类的构造函数中，使用 CountryRepository 的 GetCountries 方法创建并填充只读属性 Countries(代码文件 Templates/StyledList.xaml.cs)：

```
public ObservableCollection<Country> Countries { get; } =
  new ObservableCollection<Country>();

public StyledListBox()
{
  this.InitializeComponent();
  this.DataContext = this;
  var countries = new CountryRepository().GetCountries();
  foreach (var country in countries)
  {
    Countries.Add(country);
  }
}
```

在 XAML 代码中，定义了 countryList1 列表视图。countryList1 只使用元素的默认外观。把 ItemsSource 属

性设置为 Binding 标记扩展，它由数据绑定使用。从代码隐藏文件中，可以看到数据绑定用于一个 Country 对象数组。

图 35-17 显示了 ListView 的默认外观。在默认情况下，只在一个简单的列表中显示 ToString()方法返回的国家名称(代码文件 Templates/StyledList.xaml)。

```
<Grid>
  <ListView ItemsSource="{Binding Countries}" Margin="10"
    x:Name="countryList1" />
</Grid>
```




图 35-17

## 35.7.4 ListView 项的数据模板

接下来，使用之前为 ListView 控件创建的 DataTemplate。DataTemplate 可以直接分配给 ItemTemplate 属性(代码文件 Templates/StyledList.xaml)：

```
<ListView ItemsSource="{Binding Countries}" Margin="10"
ItemTemplate="{StaticResource CountryDataTemplate}" />
```

有了这些 XAML，项就如图 35-18 所示。

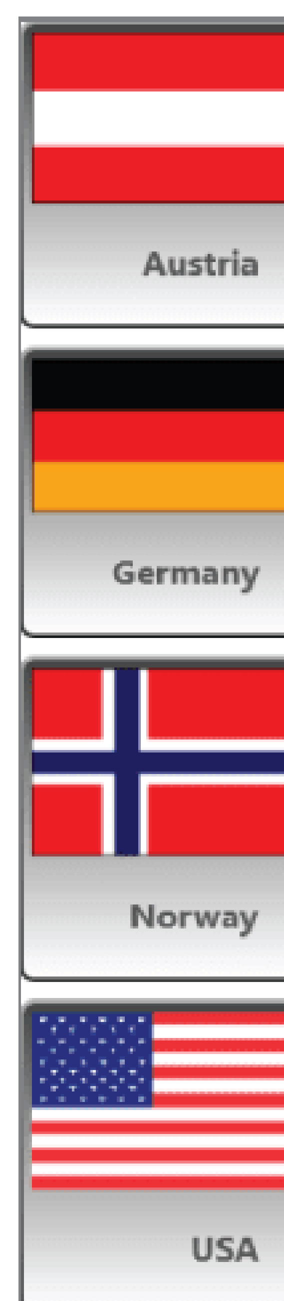



图 35-18

当然也可以定义一个引用数据模板的样式 (代码文件 Templates/Styles/ListTemplates. xaml)：

```
<Style x:Key="ListViewStyle1" TargetType="ListView">
  <Setter Property="ItemTemplate"
    Value="{StaticResource CountryDataTemplate}" />
</Style>
```

在 ListView 控件中使用这个样式(代码文件 Templates/StyledList.xaml )：

```
<ListView ItemsSource="{Binding Countries}" Margin="10"
  Style="{StaticResource ListViewStyle1}" />
```

## 35.7.5 项容器的样式

数据模板定义了每一项的外观，每项还有一个容器。ItemContainerStyle 可以定义每项的容器的外观，例如，选择、按下每个项时，应给画笔使用什么前景和背景等。对于容器边界的简单视图，设置 Margin 和 Background 属性(代码文件 Templates/Styles/ListTemplates.xaml)：

```
<Style x:Key="ListViewItemStyle1" TargetType="ListViewItem">
  <Setter Property="Background" Value="Orange"/>
  <Setter Property="Margin" Value="5" />
  <Setter Property="Template">
    <Setter.Value>
      <ControlTemplate TargetType="ListViewItem">
        <ListViewItemPresenter ContentMargin="{TemplateBinding Padding}"
          FocusBorderBrush=
            "{ThemeResource SystemControlForegroundAltHighBrush}"
          HorizontalContentAlignment=
            "{TemplateBinding HorizontalContentAlignment}"
          PlaceholderBackground=
            "{ThemeResource ListViewItemPlaceholderBackgroundThemeBrush}"
          SelectedPressedBackground=
            "{ThemeResource SystemControlHighlightListAccentHighBrush}"
          SelectedForeground=
            "{ThemeResource SystemControlHighlightAltBaseHighBrush}"
          SelectedBackground=
            "{ThemeResource SystemControlHighlightListAccentLowBrush}"
          VerticalContentAlignment=
            "{TemplateBinding VerticalContentAlignment}"/>
      </ControlTemplate>
    </Setter.Value>
  </Setter>
</Style>
```




图 35-19

样式与 ListView 的 ItemContainerStyle 属性相关联。这种样式的结果如图 35-19

所示。这个图很好地显示了项容器的边界(代码文件 Templates/StyledList.xaml)：

```
<ListView ItemsSource="{Binding Countries}" Margin="10"
  ItemContainerStyle="{StaticResource ListViewItemStyle1}"
  Style="{StaticResource ListViewStyle1}" MaxWidth="180" />
```

### 35.7.6　项面板

默认情况下， ListView 的项垂直放置。这不是在这个视图中安排项的唯一方法，还可以用其他方式安排它们，如水平放置。在项控件中安排项由项面板负责。

下面的代码片段为 ItemsPanelTemplate 定义了资源，水平布置 ItemsStackPanel，而不是垂直布置(代码文件 Templates/Styles/listTemplates.xaml)：

```
<ItemsPanelTemplate x:Key="ItemsPanelTemplate1">
  <ItemsStackPanel Orientation="Horizontal" Background="Yellow" />
</ItemsPanelTemplate>
```

下面的 ListView 声明使用与之前相同的 Style 和 ItemContainerStyle，但添加了 ItemsPanel 的资源。图 35-20 显示，项现在水平布置(代码文件 Templates/StyledList.xaml)：

```
<ItemsPanelTemplate x:Key="ItemsPanelTemplate1">
  <VirtualizingStackPanel IsItemsHost="True" Orientation="Horizontal"
    Background="Yellow"/>
</ItemsPanelTemplate>
<ListView ItemsSource="{Binding Countries}" Margin="10"
  ItemContainerStyle="{StaticResource ListViewItemStyle1}"
  Style="{StaticResource ListViewStyle1}"
  ItemsPanel="{StaticResource ItemsPanelTemplate1}" />
```




图 35-20

### 35.7.7　列表视图的控件模板

该控件还没有介绍的是滚动功能，以防项不适合放在屏幕上。定义 ListView 控件的模板可以改变这个行为。

样式 ListViewStyle2 将根据需要定义水平和垂直滚动条的行为，且项水平布置。这个样式还包括对日期模板的资源引用和前面定义的容器项模板。设置 Template 属性，现在还可以更改整个 ListView 控件的 UI (代码文件 Templates/Styles/ListTemplates.xaml)：

```
<Style x:Key="ListViewStyle2" TargetType="ListView">
  <Setter Property="ScrollViewer.HorizontalScrollBarVisibility" Value="Auto"/>
  <Setter Property="ScrollViewer.VerticalScrollBarVisibility"
    Value="Disabled"/>
  <Setter Property="ScrollViewer.HorizontalScrollMode" Value="Auto"/>
  <Setter Property="ScrollViewer.IsHorizontalRailEnabled" Value="False"/>
  <Setter Property="ScrollViewer.VerticalScrollMode" Value="Disabled"/>
  <Setter Property="ScrollViewer.IsVerticalRailEnabled" Value="False"/>
  <Setter Property="ScrollViewer.ZoomMode" Value="Disabled"/>
  <Setter Property="ScrollViewer.IsDeferredScrollingEnabled" Value="False"/>
  <Setter Property="ScrollViewer.BringIntoViewOnFocusChange" Value="True"/>
  <Setter Property="ItemTemplate"
    Value="{StaticResource CountryDataTemplate}" />
  <Setter Property="ItemContainerStyle"
    Value="{StaticResource ListViewItemStyle1}" />
  <Setter Property="ItemsPanel">
    <Setter.Value>
      <ItemsPanelTemplate>
        <ItemsStackPanel Orientation="Horizontal" Background="Yellow"/>
      </ItemsPanelTemplate>
    </Setter.Value>
  </Setter>
```

```
    <Setter Property="Template">
      <Setter.Value>
        <ControlTemplate TargetType="ListView">
          <Border BorderBrush="{TemplateBinding BorderBrush}"
            BorderThickness="{TemplateBinding BorderThickness}"
            Background="{TemplateBinding Background}">
            <ScrollViewer x:Name="ScrollViewer">
              <!-ScrollViewer definitions removed for clarity->
              <ItemsPresenter FooterTransitions=
                "{TemplateBinding FooterTransitions}"
                FooterTemplate="{TemplateBinding FooterTemplate}"
                Footer="{TemplateBinding Footer}"
                HeaderTemplate="{TemplateBinding HeaderTemplate}"
                Header="{TemplateBinding Header}"
                HeaderTransitions="{TemplateBinding HeaderTransitions}"
                Padding="{TemplateBinding Padding}"/>
            </ScrollViewer>
          </Border>
        </ControlTemplate>
      </Setter.Value>
    </Setter>
  </Style>
```

有了这个资源，ListView 的定义就很简单了，因为只需要引用 ListViewStyle2 和 ItemsSource 来检索数据(代码文件 Templates/StyledList.xaml)：

```
  <ListView ItemsSource="{Binding Countries}"
Margin="10"
    Style="{StaticResource ListViewStyle2}" />
```

新视图如图 35-21 所示。现在滚动条可用了。




图 35-21

# 35.8 动画

在动画中，可以使用移动的元素、颜色变化、变换等制作平滑的变换效果。XAML 使动画的制作非常简单。还可以连续改变任意依赖属性的值。不同的动画类可以根据其类型，连续改变不同属性的值。

动画最重要的元素是时间轴，它定义了值随时间的变化方式。有不同类型的时间轴，可用于改变不同类型的值。所有时间轴的基类都是 Timeline。为了连续改变 double 值，可以使用 DoubleAnimation 类。Int32Animation 类是 int 值的动画类。PointAnimation 类用于连续改变点，ColorAnimation 类用于连续改变颜色。

Storyboard 类可以用于合并时间轴。Storyboard 类派生自基类 TimelineGroup，TimelineGroup 又派生自基类 Timeline。

## 35.8.1 时间轴

Timeline 定义了值随时间的变化方式。下面的示例连续改变椭圆的大小。在接下来的代码中，DoubleAnimation 时间轴缩放和平移椭圆，ColorAnimation 改变填充画笔的颜色。Ellipse 类的 Triggers 属性设置为 EventTrigger。加载椭圆时触发事件。BeginStoryboard 是启动故事板的触发器动作。在故事板中，DoubleAnimation 元素用于连续改变 CompositeTransform 类的 ScaleX、ScaleY、TranslateX、TranslateY 属性。动画在 10 秒内把水平比例改为 5，垂直比例改为 3(代码文件 Animation/SimpleAnimation.xaml)：

```
  <Ellipse x:Name="ellipse1" Width="100" Height="40"
    HorizontalAlignment="Left" VerticalAlignment="Top">
    <Ellipse.Fill>
      <SolidColorBrush Color="Green" />
    </Ellipse.Fill>
    <Ellipse.RenderTransform>
      <CompositeTransform ScaleX="1" ScaleY="1" TranslateX="0" TranslateY="0" />
    </Ellipse.RenderTransform>
    <Ellipse.Triggers>
      <EventTrigger>
        <BeginStoryboard>
          <Storyboard x:Name="MoveResizeStoryboard">
            <DoubleAnimation Duration="0:0:10" To="5"
```

```
          Storyboard.TargetName="ellipse1"
          Storyboard.TargetProperty=
            "(UIElement.RenderTransform).(CompositeTransform.ScaleX)" />
        <DoubleAnimation Duration="0:0:10" To="3"
          Storyboard.TargetName="ellipse1"
          Storyboard.TargetProperty=
            "(UIElement.RenderTransform).(CompositeTransform.ScaleY)" />
        <DoubleAnimation Duration="0:0:10" To="400"
          Storyboard.TargetName="ellipse1"
          Storyboard.TargetProperty=
          "(UIElement.RenderTransform).(CompositeTransform.TranslateX)" />
        <DoubleAnimation Duration="0:0:10" To="200"
          Storyboard.TargetName="ellipse1"
          Storyboard.TargetProperty=
            "(UIElement.RenderTransform).(CompositeTransform.TranslateY)" />
        <ColorAnimation Duration="0:0:10" To="Red"
          Storyboard.TargetName="ellipse1"
          Storyboard.TargetProperty=
            "(Ellipse.Fill).(SolidColorBrush.Color)" />
      </Storyboard>
    </BeginStoryboard>
  </EventTrigger>
 </Ellipse.Triggers>
</Ellipse>
```

使用 ScaleTransform 和 TranslateTransform，动画就会访问 TransformGroup 的集合，使用一个索引器可以访问 ScaleX、ScaleY、 X 和 Y 属性：

```
<DoubleAnimation Duration="0:0:10" To="5" Storyboard.TargetName="ellipse1"
  Storyboard.TargetProperty=
    "(UIElement.RenderTransform).Children[0].(ScaleTransform.ScaleX)" />
<DoubleAnimation Duration="0:0:10" To="3" Storyboard.TargetName="ellipse1"
  Storyboard.TargetProperty=
    "(UIElement.RenderTransform).Children[0].(ScaleTransform.ScaleY)" />
<DoubleAnimation Duration="0:0:10" To="400" Storyboard.TargetName="ellipse1"
  Storyboard.TargetProperty=
    "(UIElement.RenderTransform).Children[1].(TranslateTransform.X)" />
<DoubleAnimation Duration="0:0:10" To="200" Storyboard.TargetName="ellipse1"
  Storyboard.TargetProperty=
    "(UIElement.RenderTransform).Children[1].(TranslateTransform.Y)" />
```

除了在组合变换中使用索引器之外，也可以通过名称访问 ScaleTransform 元素。下面的代码简化了该属性的名称:

```
<DoubleAnimation Duration="0:0:10" To="5" Storyboard.TargetName="scale1"
  Storyboard.TargetProperty="(ScaleX)" />
```

图 35-22 和图 35-23 显示了具有动画效果的椭圆的两个状态。

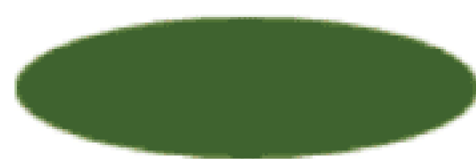


图 35-22



图 35-23

动画并不仅仅是一直和立刻显示在屏幕上的一般窗口动画，还可以给业务应用程序添加动画，使用户界面的响应性更好。光标划过按钮或单击按钮时的外观由动画定义。

Timeline 可以完成的任务如表 35-3 所示。

表 35-3

| Timeline 属性 | 说　明 |
|---|---|
| AutoReverse | 使用 AutoReverse 属性，可以指定连续改变的值在动画结束后是否返回初始值 |
| SpeedRatio | 使用 SpeedRatio，可以改变动画的移动速度。在这个属性中，可以定义父子元素的相对关系。默认值为 1；将速率设置为较小的值，会使动画移动较慢；将速率设置为高于 1 的值，会使动画移动较快 |
| BeginTime | 使用 BeginTime，可以指定从触发器事件开始到动画开始移动之间的时间长度。其单位可以是天、小时、分钟、秒和几分之秒。根据 SpeedRatio，这可以不是真实的时间。例如，如果把 SpeedRatio 设置为 2，把开始时间设置为 6 秒，动画就在 3 秒后开始 |

(续表)

| Timeline 属性 | 说　　明 |
|---|---|
| Duration | 使用 Duration 属性，可以指定动画重复一次的时间长度 |
| RepeatBehavior | 给 RepeatBehavior 属性指定一个 RepeatBehavior 结构，可以定义动画的重复次数或重复时间 |
| FillBehavior | 如果父元素的时间轴有不同的持续时间，FillBehavior 属性就很重要。例如，如果父元素的时间轴比实际动画的持续时间短，则将 FillBehavior 设置为 Stop 就表示实际动画停止。如果父元素的时间轴比实际动画的持续时间长，HoldEnd 就会一直执行动画，直到把它重置为初始值为止(假定将 AutoReverse 设置为 true) |

根据 Timeline 类的类型，还可以使用其他一些属性。例如，使用 DoubleAnimation，可以为动画的开始和结束设置 From 和 To 属性。还可以指定 By 属性，用 Bound 属性的当前值启动动画，该属性值会递增由 By 属性指定的值。

### 35.8.2　缓动函数

在前面的动画中，值以线性的方式变化。但在现实生活中，移动不会呈线性的方式。移动可能开始时较慢，逐步加快，达到最高速度，然后减缓，最后停止。一个球掉到地上，会反弹几次，最后停在地上。这种非线性行为可以使用非线性动画创建。

动画类有 EasingFunction 属性。这个属性接受一个派生自基类 EasingFunctionBase 的对象。通过这个类型，缓动函数对象可以定义值随着时间如何变化。有几个缓动函数可用于创建非线性动画，如 ExponentialEase，它给动画使用指数公式；QuadraticEase、CubicEase、QuarticEase 和 QuinticEase 的指数分别是 2、3、4、5，PowerEase 的指数是可以配置的。特别有趣的是 SineEase，它使用正弦曲线，BounceEase 创建弹跳效果，ElasticEase 用弹簧的来回震荡模拟动画值。

下面的代码把 BounceEase 函数添加到 DoubleAnimation 中。添加不同的缓动函数，就会看到动画的有趣效果：

```
<DoubleAnimation Storyboard.TargetProperty="(Ellipse.Width)"
  Duration="0:0:3" AutoReverse="True"
  FillBehavior=" RepeatBehavior="Forever"
  From="100" To="300">
  <DoubleAnimation.EasingFunction>
    <BounceEase EasingMode="EaseInOut" />
  </DoubleAnimation.EasingFunction>
</DoubleAnimation>
```

为了看到不同的缓动动画，下一个示例让椭圆在两个小矩形之间移动。Rectangle 和 Ellipse 元素在 Canvas 画布上定义，椭圆定义了 TranslateTransform 变换，来移动椭圆(代码文件 Animation/EasingFunctions.xaml)：

```
<Canvas Grid.Row="1">
  <Rectangle Fill="Blue" Width="10" Height="200" Canvas.Left="50"
    Canvas.Top="100" />
  <Rectangle Fill="Blue" Width="10" Height="200" Canvas.Left="550"
    Canvas.Top="100" />
  <Ellipse Fill="Red" Width="30" Height="30" Canvas.Left="60" Canvas.Top="185">
    <Ellipse.RenderTransform>
      <TranslateTransform x:Name="translate1" X="0" Y="0" />
    </Ellipse.RenderTransform>
  </Ellipse>
</Canvas>
```

图 35-24 显示了矩形和椭圆。

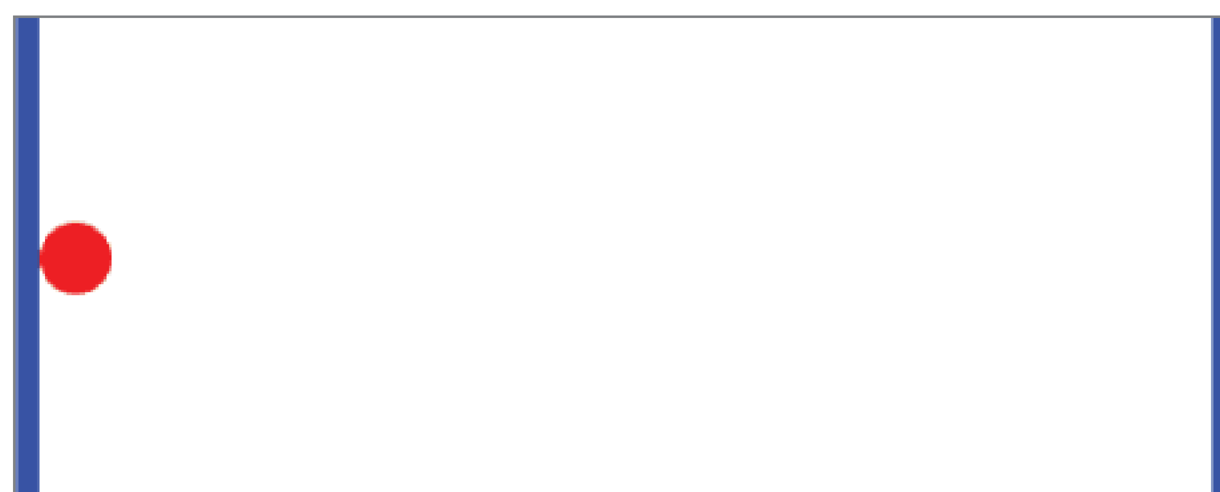


图 35-24

用户单击一个按钮，启动动画。单击此按钮之前，用户可以从 ComboBox comboEasingFunctions 中选择缓动函数，使用单选按钮选择一个 EasingMode 枚举值。

```
<StackPanel Orientation="Horizontal">
  <ComboBox x:Name="comboEasingFunctions" Margin="10" />
  <Button Click="OnStartAnimation" Margin="10">Start</Button>
  <Border BorderThickness="1" BorderBrush="Black" Margin="3">
    <StackPanel Orientation="Horizontal">
      <RadioButton x:Name="easingModeIn" GroupName="EasingMode" Content="In" />
      <RadioButton x:Name="easingModeOut" GroupName="EasingMode"
        Content="Out" IsChecked="True" />
      <RadioButton x:Name="easingModeInOut" GroupName="EasingMode"
        Content="InOut" />
    </StackPanel>
  </Border>
</StackPanel>
```

ComboBox 中显示的、动画激活的缓动函数列表从 EasingFunctionManager 的 EasingFunctionModels 属性中返回。这个管理器把缓动函数转换为 EasingFunctionModel，以显示出来(代码文件 Animation /EasingFunctionsManager.cs)：

```
public class EasingFunctionsManager
{
  private static IEnumerable<EasingFunctionBase> s_easingFunctions =
    new List<EasingFunctionBase>()
    {
      new BackEase(),
      new SineEase(),
      new BounceEase(),
      new CircleEase(),
      new CubicEase(),
      new ElasticEase(),
      new ExponentialEase(),
      new PowerEase(),
      new QuadraticEase(),
      new QuinticEase()
    };

  public IEnumerable<EasingFunctionModel> EasingFunctionModels =>
    s_easingFunctions.Select(f => new EasingFunctionModel(f));
}
```

EasingFunctionModel 类定义了 ToString 方法，返回定义了缓动函数的类的名称。这个名字显示在组合框中(代码文件 Animation/EasingFunctionModel.cs)：

```
public class EasingFunctionModel
{
  public EasingFunctionModel(EasingFunctionBase easingFunction) =>
    EasingFunction = easingFunction;

  public EasingFunctionBase EasingFunction { get; }

  public override string ToString() => EasingFunction.GetType().Name;
}
```

ComboBox 在代码隐藏文件的构造函数中填充(代码文件 Animation/EasingFunctions. xaml.cs)：

```
private EasingFunctionsManager _easingFunctions = new EasingFunctionsManager();
private const int AnimationTimeSeconds = 6;
```

```
public EasingFunctions()
{
  this.InitializeComponent();
  foreach (var easingFunctionModel in _easingFunctions.EasingFunctionModels)
  {
    comboEasingFunctions.Items.Add(easingFunctionModel);
  }
}
```

在用户界面中，不仅可以选择应该用于动画的缓动函数的类型，也可以选择缓动模式。所有缓动函数的基类(EasingFunctionBase)定义了 EasingMode 属性，它可以是 EasingMode 枚举的值。

单击此按钮，启动动画，会调用 OnStartAnimation 方法。该方法又调用 StartAnimation 方法。在这个方法中，通过编程方式创建一个包含 DoubleAnimation 的故事板。之前列出了使用 XAML 的类似代码。动画连续改变 translate1 元素的 X 属性 (代码文件 Animation/EasingFunctionsPage.xaml.cs)：

```
private void OnStartAnimation(object sender, RoutedEventArgs e)
{
  var easingFunctionModel =
    comboEasingFunctions.SelectedItem as EasingFunctionModel;
  if (easingFunctionModel != null)
  {
    EasingFunctionBase easingFunction = easingFunctionModel.EasingFunction;
    easingFunction.EasingMode = GetEasingMode();
    StartAnimation(easingFunction);
  }
}

private void StartAnimation(EasingFunctionBase easingFunction)
{
  var storyboard = new Storyboard();
  var ellipseMove = new DoubleAnimation();
  ellipseMove.EasingFunction = easingFunction;
  ellipseMove.Duration = new
    Duration(TimeSpan.FromSeconds(AnimationTimeSeconds));
  ellipseMove.From = 0;
  ellipseMove.To = 460;
  Storyboard.SetTarget(ellipseMove, translate1);
  Storyboard.SetTargetProperty(ellipseMove, "X");

  // start the animation in 0.5 seconds
  ellipseMove.BeginTime = TimeSpan.FromSeconds(0.5);

  // keep the position after the animation
  ellipseMove.FillBehavior = FillBehavior.HoldEnd;
  storyboard.Children.Add(ellipseMove);
  storyBoard.Begin();
}
```

现在，可以运行应用程序，看看椭圆使用不同的缓动函数，以不同的方式从左矩形移动到右矩形。使用一些缓动函数，比如 BackEase、BounceEase 或 ElasticEase，区别是显而易见的。其他的一些缓动函数没有明显的区别。为了更好地理解缓动值如何变化，可以创建一个折线图，其中显示了一条线，其上的值由基于时间的缓动函数返回。

为了显示折线图，可以创建一个用户控件，它定义了一个 Canvas 元素。默认情况下，x 方向从左到右，y 方向从上到下。为了把 y 方向改为从下到上，可以定义一个变换(代码文件 Animation/EasingChartControl.xaml)：

```
<Canvas x:Name="canvas1" Width="500" Height="500" Background="Yellow">
  <Canvas.RenderTransform>
    <TransformGroup>
      <ScaleTransform ScaleX="1" ScaleY="-1" />
      <TranslateTransform X="0" Y="500" />
    </TransformGroup>
  </Canvas.RenderTransform>
</Canvas>
```

在代码隐藏文件中，使用线段绘制折线图。线段在本章的 35.3.1 节“使用段的几何图形”中用 XAML 代码讨论过。这里，它们可以在代码中使用。通过传递 x 轴上显示的时间值的规范化值，缓动函数的 Ease 方法就返回一个值，显示在 y 轴上(代码文件 Animation/EasingChartControl.xaml.cs )：

```
private const double SamplingInterval = 0.01;

public void Draw(EasingFunctionBase easingFunction)
{
  canvas1.Children.Clear();
  var pathSegments = new PathSegmentCollection();
  for (double i = 0; i < 1; i += _samplingInterval)
  {
    double x = i * canvas1.Width;
    double y = easingFunction.Ease(i) * canvas1.Height;
    var segment = new LineSegment();
    segment.Point = new Point(x, y);
    pathSegments.Add(segment);
  }

  var p = new Path();
  p.Stroke = new SolidColorBrush(Colors.Black);
  p.StrokeThickness = 3;
  var figures = new PathFigureCollection();
  figures.Add(new PathFigure { Segments = pathSegments });
  p.Data = new PathGeometry { Figures = figures };
  canvas1.Children.Add(p);
}
```

EasingChartControl 的 Draw 方法在动画开始时调用 (代码文件 Animation/EasingFunctions.xaml.cs)：

```
private void StartAnimation(EasingFunctionBase easingFunction)
{
  // show the chart
  chartControl.Draw(easingFunction);
  //...
```

运行应用程序时，可以看到使用 CubicEase 和 EaseOut 的结果，如图 35-25 所示。选择 EaseIn 时，值在动画的开始变化得较慢，在动画的后面变化得较快，如图 35-26 所示。图 35-27 显示使用 CubicEase 和 EaseInOut 的效果。BounceEase、BackEase 和 ElasticEase 的图表如图 35-28、图 35-29 和图 35-30 所示。

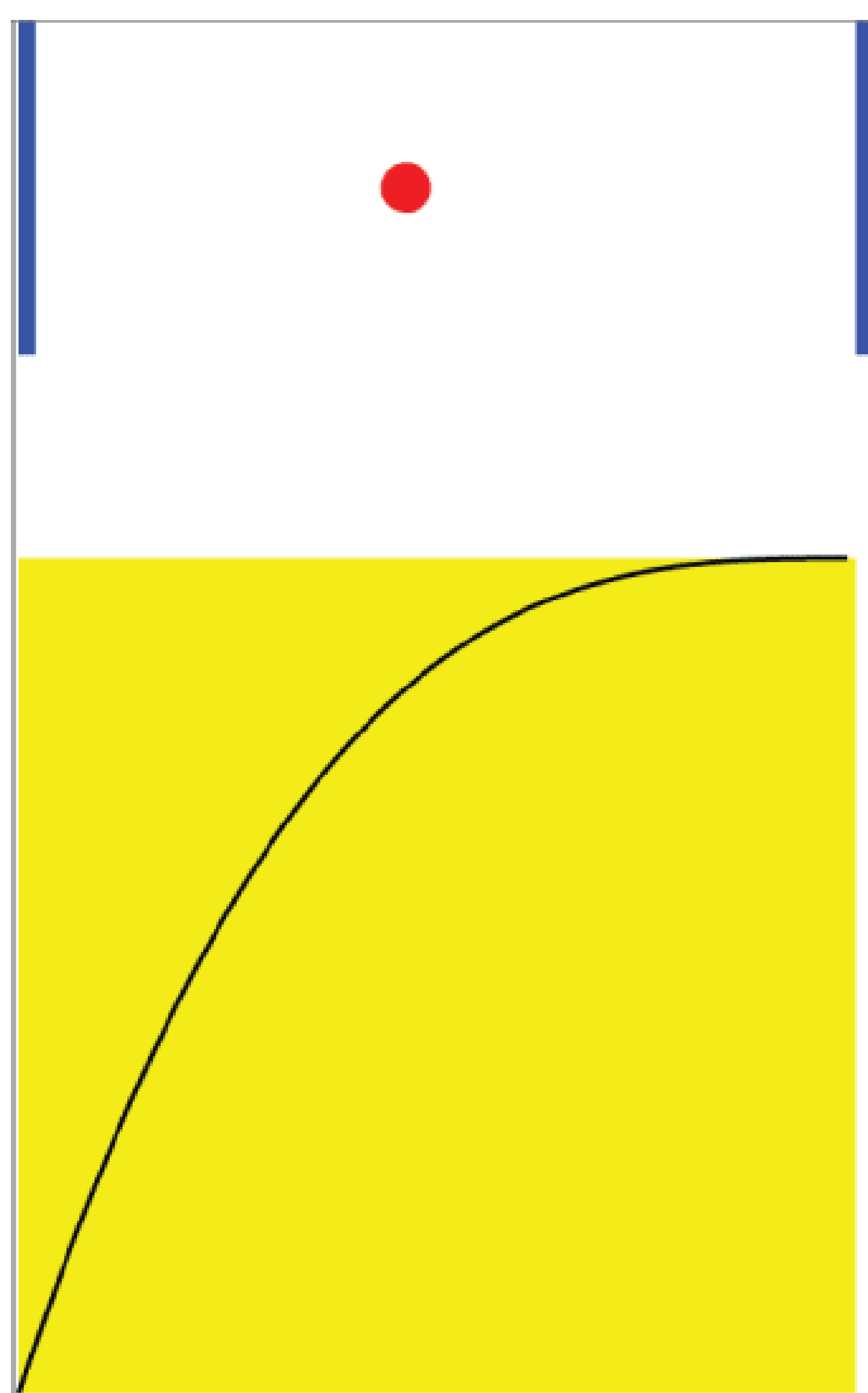


图 35-25

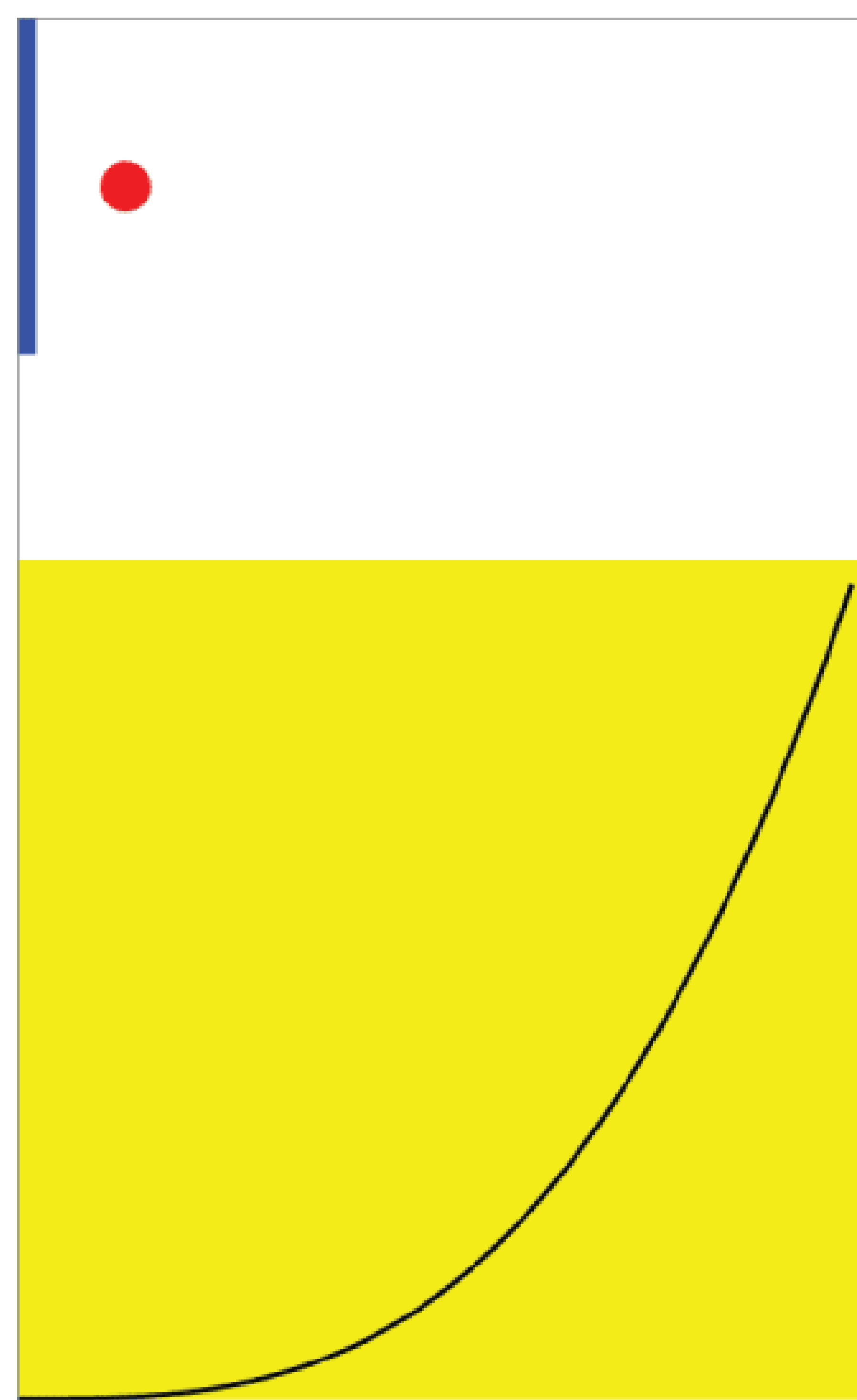


图 35-26

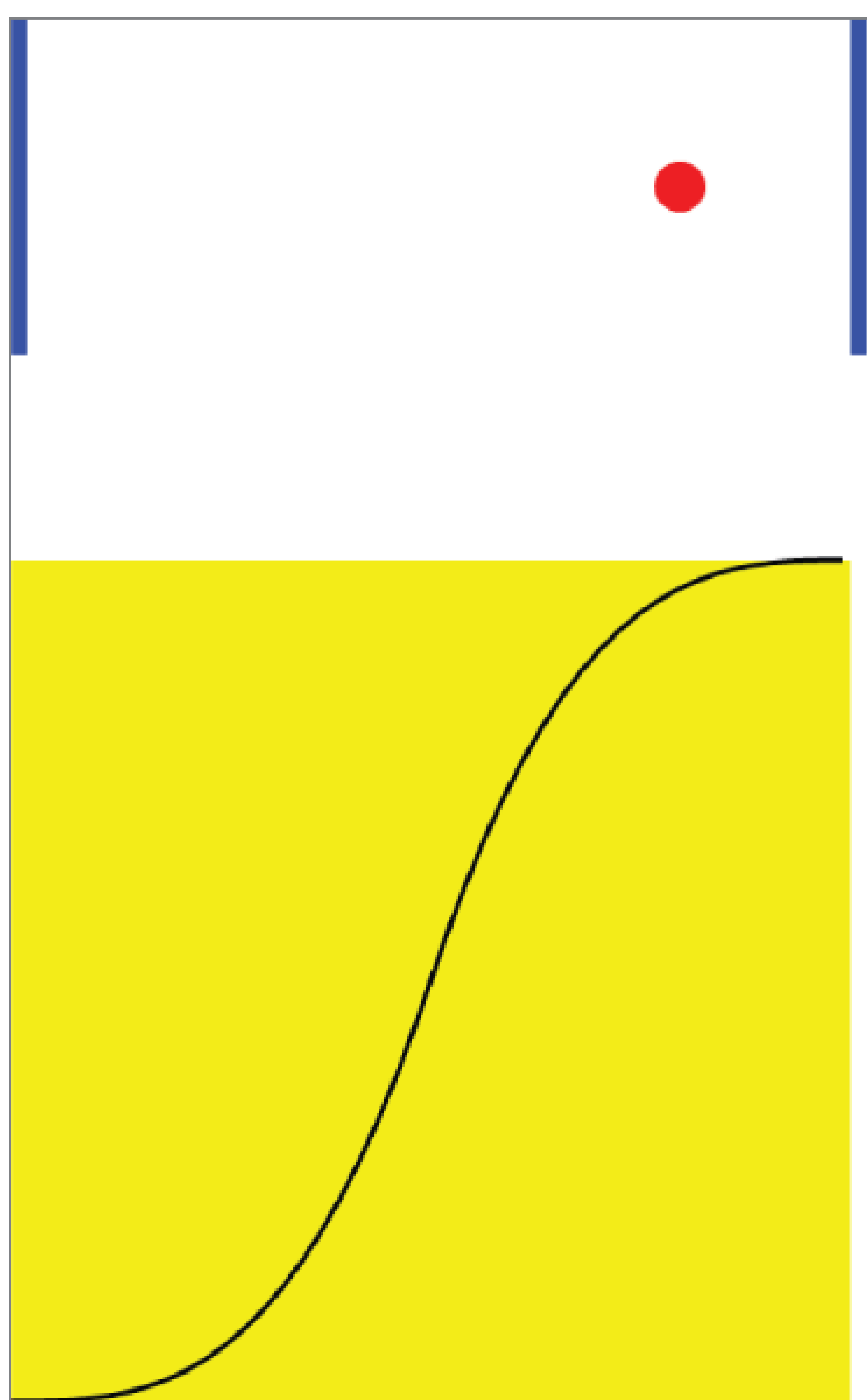


图 35-27

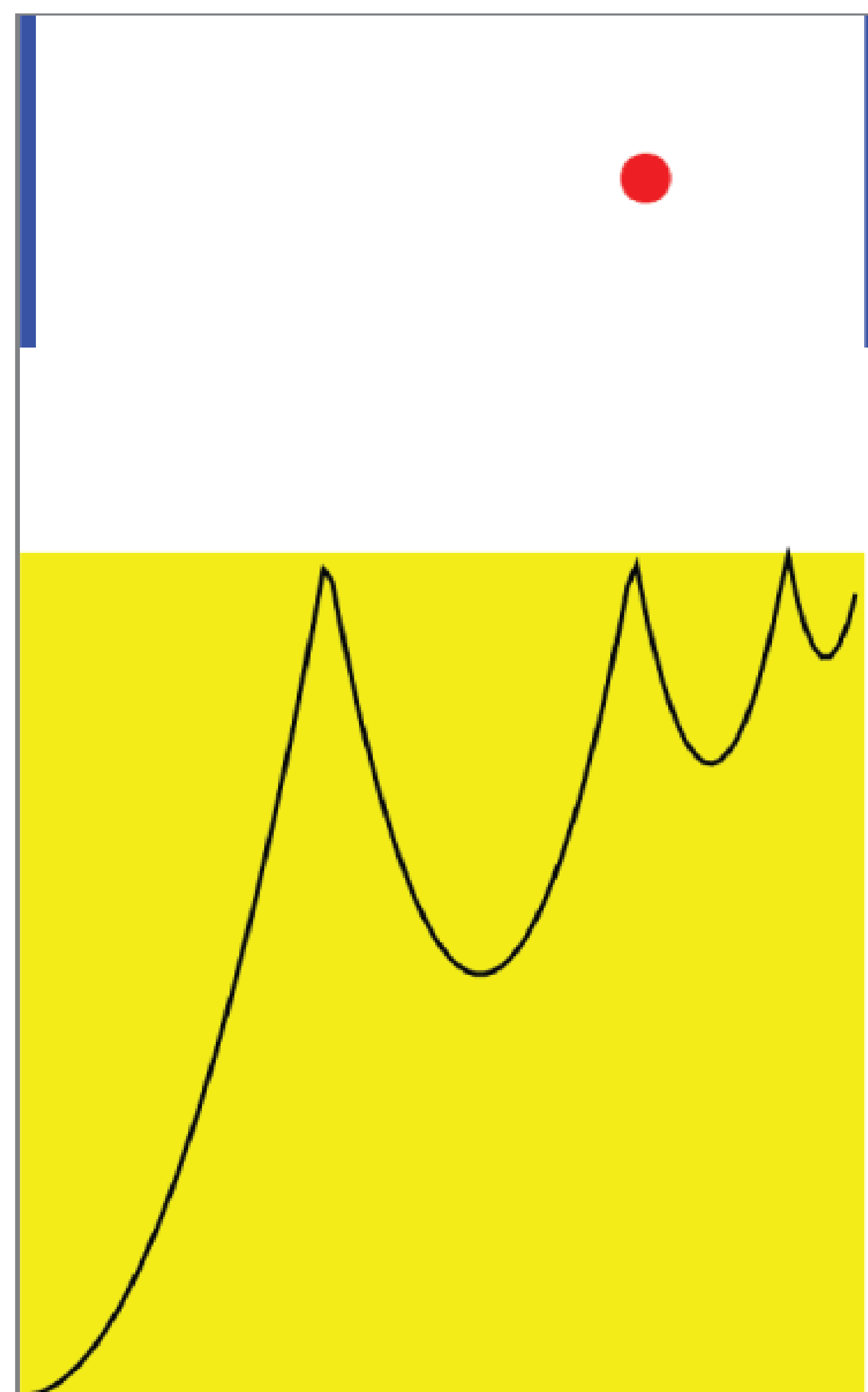


图 35-28

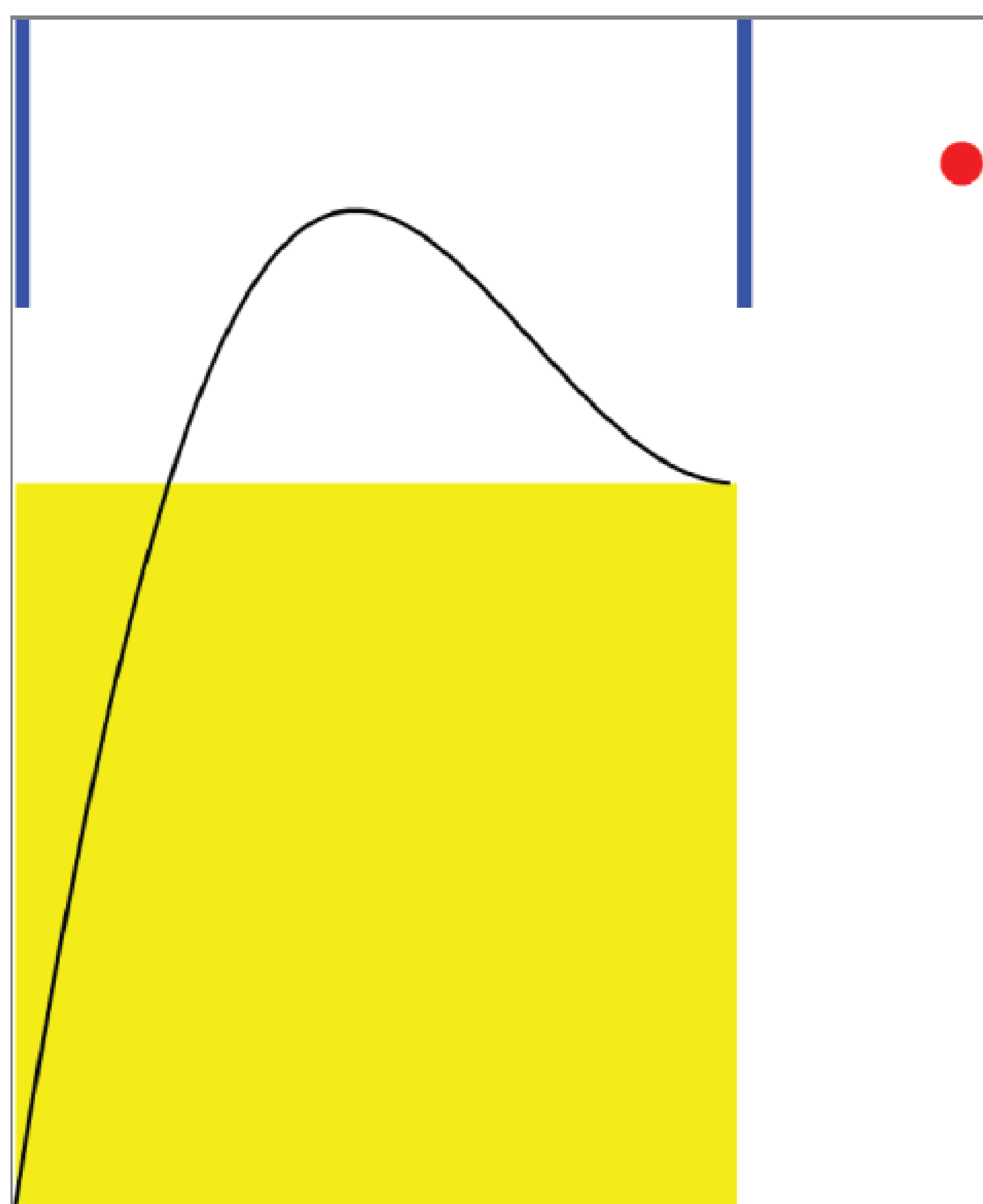


图 35-29

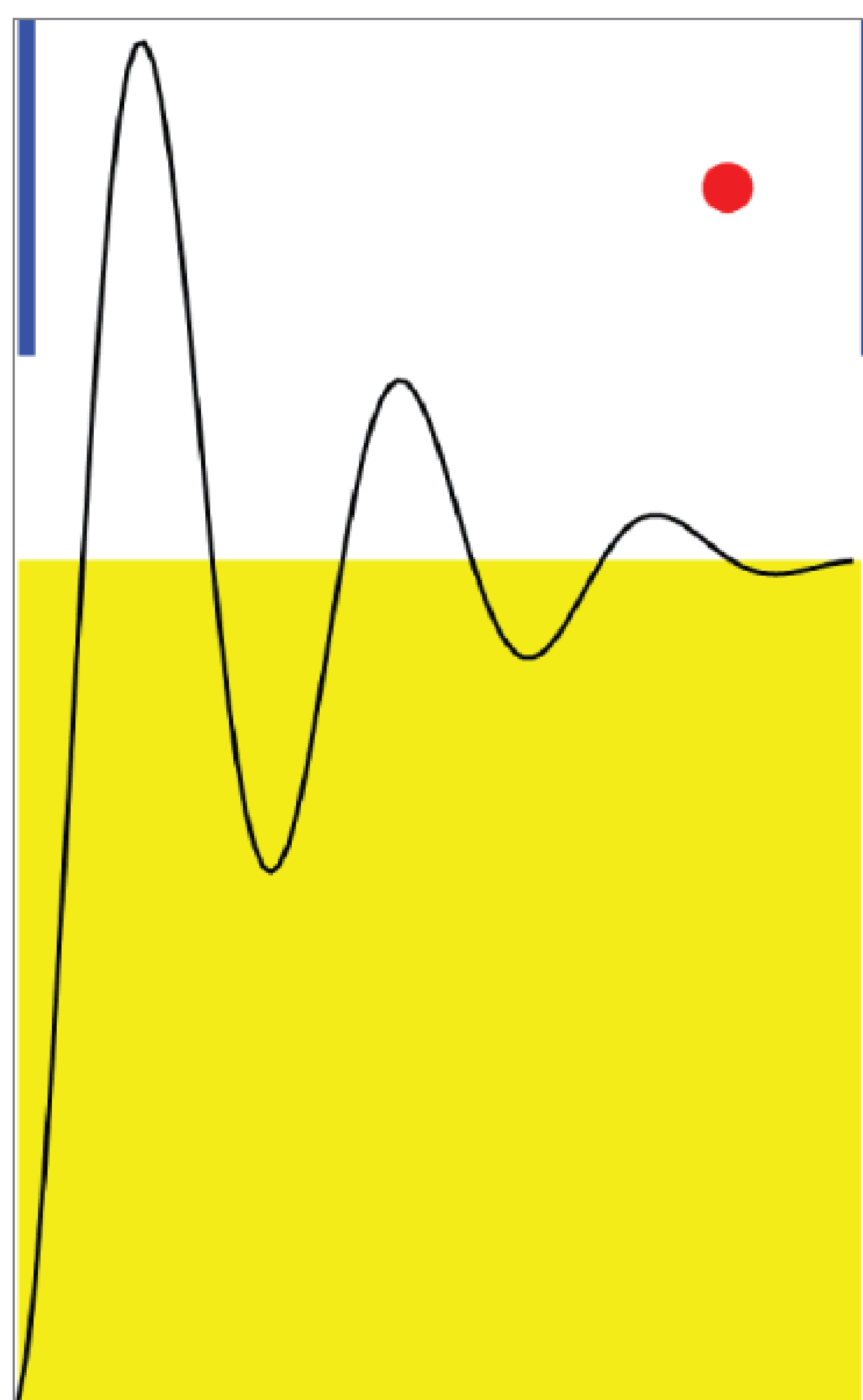


图 35-30

### 35.8.3　关键帧动画

如前所述，使用缓动函数，就可以用非线性的方式制作动画。如果需要为动画指定几个值，就可以使用关键帧动画。与正常的动画一样，关键帧动画也有不同的动画类型，它们可以改变不同类型的属性。

DoubleAnimationUsingKeyFrames 是双精度类型的关键帧动画。其他关键帧动画类型有 Int32AnimationUsingKeyFrames、PointAnimationUsingKeyFrames、ColorAnimationUsingKeyFrames、

SizeAnimationUsingKeyFrames以及ObjectAnimationUsingKeyFrames。

示例 XAML 代码连续地改变 TranslateTransform 元素的 X 值和 Y 值，从而改变椭圆的位置。把 EventTrigger 定义为 RountedEvent Ellipse.Loaded，动画就会在加载椭圆时启动。事件触发器用 BeginStoryboard 元素启动一个 Storyboard。该 Storyboard 包含两个 DoubleAnimationUsingKeyFrame 类型的关键帧动画。关键帧动画由帧元素组成。第一幅关键帧动画使用一个 LinearKeyFrame、一个 DiscreteDoubleKeyFrame 和一个 SplineDoubleKeyFrame；第二幅关键帧动画是一个 EasingDoubleKeyFrame。LinearDoubleKeyFrame 使对应值线性变化。KeyTime 属性定义了动画应何时达到 Value 属性的值。

这里 LinearDoubleKeyFrame 用 3 秒的时间使 X 属性到达值 30。DiscreteDoubleKeyFrame 在 4 秒后立即改变为新值。SplineDoubleKeyFrame 使用贝塞尔曲线，其中的两个控制点由 KeySpline 属性指定。EasingDoubleKeyFrame 是一个帧类，它支持设置缓动函数(如 BounceEase)来控制动画值(代码文件 Animation/KeyFrameAnimationPage.xaml)：

```
<Canvas>
  <Ellipse Fill="Red" Canvas.Left="20" Canvas.Top="20" Width="25" Height="25">
    <Ellipse.RenderTransform>
      <TranslateTransform X="50" Y="50" x:Name="ellipseMove" />
    </Ellipse.RenderTransform>
    <Ellipse.Triggers>
      <EventTrigger>
        <BeginStoryboard>
          <Storyboard>
            <DoubleAnimationUsingKeyFrames Storyboard.TargetProperty="X"
              Storyboard.TargetName="ellipseMove">
              <LinearDoubleKeyFrame KeyTime="0:0:2" Value="30" />
              <DiscreteDoubleKeyFrame KeyTime="0:0:4" Value="80" />
              <SplineDoubleKeyFrame KeySpline="0.5,0.0 0.9,0.0"
                KeyTime="0:0:10" Value="300" />
              <LinearDoubleKeyFrame KeyTime="0:0:20" Value="150" />
            </DoubleAnimationUsingKeyFrames>
            <DoubleAnimationUsingKeyFrames Storyboard.TargetProperty="Y"
              Storyboard.TargetName="ellipseMove">
              <SplineDoubleKeyFrame KeySpline="0.5,0.0 0.9,0.0"
                KeyTime="0:0:2" Value="50" />
              <EasingDoubleKeyFrame KeyTime="0:0:20" Value="300">
                <EasingDoubleKeyFrame.EasingFunction>
                  <BounceEase />
                </EasingDoubleKeyFrame.EasingFunction>
              </EasingDoubleKeyFrame>
            </DoubleAnimationUsingKeyFrames>
          </Storyboard>
        </BeginStoryboard>
      </EventTrigger>
    </Ellipse.Triggers>
  </Ellipse>
</Canvas>
```

## 35.8.4　过渡

为方便创建带动画的用户界面，UWP 应用程序定义了过渡效果。过渡效果更容易创建引人注目的应用程序，而不需要考虑如何制作很酷的动画。过渡效果预定义了如下动画：添加、移除和重新排列列表上的项，打开面板，改变内容控件的内容等。

下面的示例演示了几个过渡效果，在用户控件的左边和右边展示它们，再显示没有过渡效果的相似元素，这有助于看到它们之间的差异。当然，需要启动应用程序才能看到区别，很难在印刷出来的书上证明这一点。

### 1. 复位过渡效果

第一个例子在按钮元素的 Transitions 属性中使用了 RepositionThemeTransition。过渡效果总是需要在 TransitionCollection 内定义，因为这样的集合是不会自动创建的，如果没有使用 TransitionCollection，就会显示一个有误导作用的运行时错误。第二个按钮不使用过渡效果(代码文件 Transitions/RepositionUserControl.xaml)：

```
<Button Grid.Row="1" Click="OnReposition" Content="Reposition"
  x:Name="buttonReposition" Margin="10">
```

```
    <Button.Transitions>
      <TransitionCollection>
        <RepositionThemeTransition />
      </TransitionCollection>
    </Button.Transitions>
  </Button>
  <Button Grid.Row="1" Grid.Column="1" Click="OnReset" Content="Reset"
    x:Name="button2" Margin="10" />
```

RepositionThemeTransition 是控件改变其位置时的过渡效果。在代码隐藏文件中，用户单击按钮时，Margin 属性会改变，按钮的位置也会改变。

```
private void OnReposition(object sender, RoutedEventArgs e)
{
  buttonReposition.Margin = new Thickness(100);
  button2.Margin = new Thickness(100);
}

private void OnReset(object sender, RoutedEventArgs e)
{
  buttonReposition.Margin = new Thickness(10);
  button2.Margin = new Thickness(10);
}
```

### 2. 窗格过渡效果

PopupThemeTransition 和 PaneThemeTransition 显示在下一个用户控件中。在这里，过渡效果用 Popup 控件的 ChildTransitions 属性定义(代码文件 Transitions /PaneTransitionUserControl.xaml)：

```
<StackPanel Orientation="Horizontal" Grid.Row="2">
  <Popup x:Name="popup1" Width="200" Height="90" Margin="60">
    <Border Background="Red" Width="100" Height="60">
    </Border>
    <Popup.ChildTransitions>
      <TransitionCollection>
        <PopupThemeTransition />
      </TransitionCollection>
    </Popup.ChildTransitions>
  </Popup>
  <Popup x:Name="popup2" Width="200" Height="90" Margin="60">
    <Border Background="Red" Width="100" Height="60">
    </Border>
    <Popup.ChildTransitions>
      <TransitionCollection>
        <PaneThemeTransition />
      </TransitionCollection>
    </Popup.ChildTransitions>
  </Popup>
  <Popup x:Name="popup3" Margin="60" Width="200" Height="90">
    <Border Background="Green" Width="100" Height="60">
    </Border>
  </Popup>
</StackPanel>
```

代码隐藏文件通过设置 IsOpen 属性，打开和关闭 Popup 控件。这会启动过渡效果(代码文件 Transitions\PaneTransitionUserControl.xaml)：

```
private void OnShow(object sender, RoutedEventArgs e)
{
  popup1.IsOpen = true;
  popup2.IsOpen = true;
  popup3.IsOpen = true;
}

private void OnHide(object sender, RoutedEventArgs e)
{
  popup1.IsOpen = false;
  popup2.IsOpen = false;
  popup3.IsOpen = false;
}
```

运行应用程序时可以看到，打开 Popup 和 Flyout 控件的 PopupThemeTransition 看起来不错。PaneThemeTransition 慢慢从右侧打开 Popup。这个过渡效果也可以通过设置属性，配置为从其他侧边打开，因

此最适合面板，例如设置栏，它从一个侧边移入。

### 3. 项的过渡效果

从项控件中添加和删除项也定义了过渡效果。以下的 ItemsControl 利用了 EntranceThemeTransition 和 RepositionThemeTransition。项添加到集合中时使用 EntranceThemeTransition，重新安排项时，例如从列表中删除项时，使用 RepositionThemeTransition (代码文件 Transitions/ListItemsUserControl.xaml)：

```
<ItemsControl Grid.Row="1" x:Name="list1">
  <ItemsControl.ItemContainerTransitions>
    <TransitionCollection>
      <EntranceThemeTransition />
      <RepositionThemeTransition />
    </TransitionCollection>
  </ItemsControl.ItemContainerTransitions>
</ItemsControl>
<ItemsControl Grid.Row="1" Grid.Column="1" x:Name="list2" />
```

在代码隐藏文件中，Rectangle 对象在列表控件中添加和删除。一个 ItemsControl 对象没有关联的过渡效果，所以运行应用程序时，很容易看出差异(代码文件 Transitions/ListItemsUserControl. xaml.cs)：

```
private void OnAdd(object sender, RoutedEventArgs e)
{
  list1.Items.Add(CreateRectangle());
  list2.Items.Add(CreateRectangle());
}

private Rectangle CreateRectangle() =>
  new Rectangle
  {
    Width = 90,
    Height = 40,
    Margin = new Thickness(5),
    Fill = new SolidColorBrush { Color = Colors.Blue }
  };

private void OnRemove(object sender, RoutedEventArgs e)
{
  if (list1.Items.Count > 0)
  {
    list1.Items.RemoveAt(0);
    list2.Items.RemoveAt(0);
  }
}
```

**注意：**

通过这些过渡效果，了解了如何减少使用户界面连续动起来所需的工作量。一定要查看可用于 UWP 应用程序的更多过渡效果。查看 MSDN 文档的 Transition 中的派生类，可以看到所有的过渡效果。

## 35.9　可视化状态管理器

本章前面的“控件模板”中，介绍了如何创建控件模板，自定义控件的外观。其中还缺了些什么。使用按钮的默认模板，按钮会响应鼠标的移动和单击，当鼠标移动到按钮或单击按钮时，按钮的外观是不同的。这种外观变化通过可视化状态和动画来处理，由可视化状态管理器控制。

本节介绍如何改变按钮样式，来响应鼠标的移动和单击，还描述了如何创建自定义状态，当几个控件应该切换到禁用状态时，例如进行一些后台处理时，这些自定义状态用于处理完整页面的变化。

对于 XAML 元素，可以定义可视化状态、状态组和状态，指定状态的特定动画。状态组允许同时有多个状态。对于一组，一次只能有一个状态。然而，另一组的另一个状态可以在同一时间激活。例如，按钮的状态和状态组。按钮控件定义了状态组 CommonStates 和 FocusStates。用 FocusStates 定义的状态是 Focused、Unfocused 和 PointerFocused， CommonStates 组定义了状态 Normal、PointerOver、Pressed 和 Disabled。有了这些选项，多个状态可以同时激活，但一个状态组内总是只有一个状态是激活的。例如，按钮可以是 Focused 和 Normal

状态。它也可以是 Focused 和 Pressed 状态，还可以定义定制的状态和状态组。

下面看看具体的例子。

### 35.9.1 用控件模板预定义状态

下面利用先前创建的自定义控件模板，样式化按钮控件，使用可视化状态改进它。为此，一个简单的方法是使用 Microsoft Blend for Visual Studio。图 35-31 显示了状态窗口，选择控件模板时就会显示该窗口。在这里可以看到控件的可用状态，并基于这些状态记录变化。

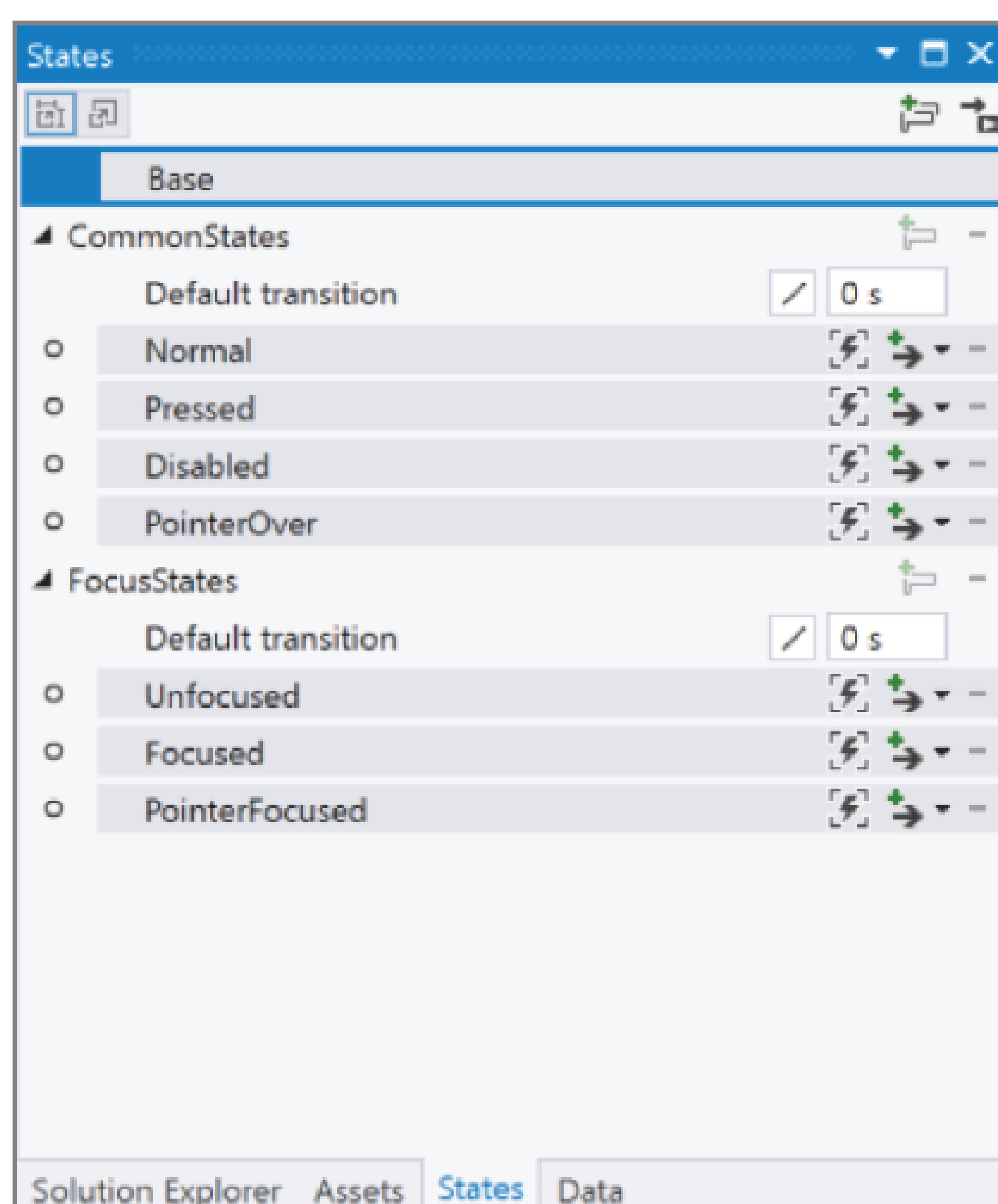



图 35-31

之前的按钮模板改为定义可视化状态：Pressed、Disabled 和 PointerOver。在状态中，Storyboard 定义了一个 ColorAnimation 来改变椭圆的 Fill 属性的颜色(代码文件 VisualStates/MainPage.xaml)：

```
<Style x:Key="RoundedGelButton" TargetType="Button">
  <Setter Property="Width" Value="100" />
  <Setter Property="Height" Value="100" />
  <Setter Property="Foreground" Value="White" />
  <Setter Property="Template">
    <Setter.Value>
      <ControlTemplate TargetType="Button">
        <Grid>
          <VisualStateManager.VisualStateGroups>
            <VisualStateGroup x:Name="CommonStates">
              <VisualState x:Name="Normal"/>
              <VisualState x:Name="Pressed">
                <Storyboard>
                  <ColorAnimation Duration="0" To="#FFC8CE11"
                    Storyboard.TargetProperty=
                      "(Shape.Fill).(SolidColorBrush.Color)"
                    Storyboard.TargetName=
                      "GelBackground" />
                </Storyboard>
              </VisualState>
              <VisualState x:Name="Disabled">
                <Storyboard>
                  <ColorAnimation Duration="0" To="#FF606066"
                    Storyboard.TargetProperty=
                      "(Shape.Fill).(SolidColorBrush.Color)"
                    Storyboard.TargetName="GelBackground" />
                </Storyboard>
              </VisualState>
              <VisualState x:Name="PointerOver">
                <Storyboard>
                  <ColorAnimation Duration="0" To="#FF0F9D3A"
                    Storyboard.TargetProperty=
```

```
                "(Shape.Fill).(SolidColorBrush.Color)"
              Storyboard.TargetName="GelBackground" />
          </Storyboard>
        </VisualState>
      </VisualStateGroup>
    </VisualStateManager.VisualStateGroups>
    <Ellipse x:Name="GelBackground" StrokeThickness="0.5" Fill="Black">
      <Ellipse.Stroke>
      <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
        <GradientStop Offset="0" Color="#ff7e7e7e" />
        <GradientStop Offset="1" Color="Black" />
      </LinearGradientBrush>
    </Ellipse.Stroke>
    </Ellipse>
    <Ellipse Margin="15,5,15,50">
      <Ellipse.Fill>
        <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
          <GradientStop Offset="0" Color="#aaffffff" />
          <GradientStop Offset="1" Color="Transparent" />
        </LinearGradientBrush>
      </Ellipse.Fill>
    </Ellipse>
    <ContentPresenter x:Name="GelButtonContent"
      VerticalAlignment="Center"
      HorizontalAlignment="Center"
      Content="{TemplateBinding Content}" />
  </Grid>
</ControlTemplate>
</Setter.Value>
</Setter>
</Style>
```

现在运行应用程序，可以看到颜色随着鼠标的移动和单击而变化。

### 35.9.2　定义自定义状态

使用 VisualStateManager 可以定义定制的状态，使用 VisualStateGroup 和 VisualState 的状态可以定义定制的状态组。下面的代码片段在 CustomStates 组内创建了 Enabled 和 Disabled 状态。可视化状态在主窗口的网格中定义。改变状态时，Button 元素的 IsEnabled 属性使用 DiscreteObjectKeyFrame 动画立即改变（代码文件 VisualStates/MainPage.xaml）：

```
<VisualStateManager.VisualStateGroups>
  <VisualStateGroup x:Name="CustomStates">
    <VisualState x:Name="Enabled"/>
    <VisualState x:Name="Disabled">
      <Storyboard>
        <ObjectAnimationUsingKeyFrames
          Storyboard.TargetProperty="(Control.IsEnabled)"
          Storyboard.TargetName="button1">
          <DiscreteObjectKeyFrame KeyTime="0">
            <DiscreteObjectKeyFrame.Value>
              <x:Boolean>False</x:Boolean>
            </DiscreteObjectKeyFrame.Value>
          </DiscreteObjectKeyFrame>
        </ObjectAnimationUsingKeyFrames>
        <ObjectAnimationUsingKeyFrames
          Storyboard.TargetProperty="(Control.IsEnabled)"
          Storyboard.TargetName="button2">
          <DiscreteObjectKeyFrame KeyTime="0">
            <DiscreteObjectKeyFrame.Value>
              <x:Boolean>False</x:Boolean>
            </DiscreteObjectKeyFrame.Value>
          </DiscreteObjectKeyFrame>
        </ObjectAnimationUsingKeyFrames>
      </Storyboard>
    </VisualState>
  </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

### 35.9.3　设置自定义的状态

现在需要设置状态。为此，可以调用 VisualStateManager 类的 GoToState 方法。在代码隐藏文件中，OnEnable

和 OnDisable 方法是页面上两个按钮的 Click 事件处理程序(代码文件 VisualStates/MainPage.xaml.cs)：

```
private void OnEnable(object sender, RoutedEventArgs e)
{
  VisualStateManager.GoToState(this, "Enabled", useTransitions: true);
}

private void OnDisable(object sender, RoutedEventArgs e)
{
  VisualStateManager.GoToState(this, "Disabled", useTransitions: true);
}
```

在真实的应用程序中，可以以类似的方式更改状态，例如执行网络调用时，用户不应该处理页面内的一些控件。用户仍应被允许单击取消按钮。通过改变状态，还可以显示进度信息。

## 35.10 小结

本章介绍了样式化 Windows 应用程序的许多功能。XAML 便于分开开发人员和设计人员的工作。所有 UI 功能都可以使用 XAML 创建，其功能用代码隐藏文件创建。

我们还探讨了许多形状和几何图形元素，它们是后面几章学习的所有其他控件的基础。基于矢量的图形允许 XAML 元素缩放、剪切和旋转。

可以使用不同类型的画笔绘制背景和前景元素，不仅可以使用纯色画笔、线性渐变或放射性渐变画笔，而且可以使用可视化画笔完成反射功能或显示视频。

样式和模板可以定制控件的外观。可视化状态管理器可以动态更改 XAML 元素的属性。连续改变 WPF 控件的属性值，就可以轻松地制作出动画。第 36 章将继续介绍 Windows 应用程序，深入论述 UWP 的高级功能。

# 第 36 章 高级 Windows 应用程序

**本章要点**

- 共享数据
- 使用应用程序服务
- 编译的绑定功能
- 富文本显示
- 使用墨水
- AutoSuggest

**本章源代码下载地址(wrox.com)：**

打开 www.wrox.com 的 Download Code 选项卡可下载本章源代码。源代码也可以在 AdvancedWindows 目录的 https://github.com/ProfessionalCSharp/ProfessionalCSharp7 中找到。本章的代码只包含一个大示例，它展示了本章的各个方面：

- AppLifetime
- Sharing Samples
- AppServices
- CompiledBindingLifetime
- CompiledBindingMethods
- PhasedBinding
- TextSample
- Text Overflow
- InkSample
- AutoSuggestSample

## 36.1 概述

前一章介绍了 Windows 应用程序的用户界面(UI)元素、共享代码的模式和用 XAML 样式化应用程序。本章继续讨论 Windows 应用程序特定的几个方面，Windows 应用程序的生命周期管理不同于传统的桌面应用程序，

用共享协定创建共享源和目标应用程序，在应用程序之间共享数据。使用带有编译绑定的高级绑定特性，创建文本流，并使用 AutoSuggestBox 自动完成用户输入。

下面先讨论 Windows 应用程序的生命周期，它不同于传统的桌面应用程序的生命周期。

# 36.2 应用程序的生命周期

Windows 8 为应用程序引入了一个新的生命周期，完全不同于传统的桌面应用程序的生命周期。在 Windows 8.1 中有些变化，在 Windows 10 中又有一些变化。如果使用 Windows10 和平板电脑模式，应用程序的生命周期与桌面模式是不同的。在平板电脑模式中，应用程序通常全屏显示。分离键盘(对于平板电脑设备，如 Microsoft Surface)，或在 Action Center 中使用 Tablet Mode 按钮，可以自动切换到平板模式。在平板模式下运行应用程序时，如果应用程序进入后台(用户切换到另一个应用程序)，就会暂停，它不会得到任何更多的 CPU 利用率。这样，应用程序不消耗任何电力。应用程序在后台时，只使用内存，一旦用户切换到这个应用程序，应用程序就再次激活。

当内存资源短缺时，Windows 可以终止暂停应用程序的进程，从而终止该应用程序。应用程序不会收到任何消息，所以不能对此事件做出反应。因此，应用程序应该在进入暂停模式前做一些处理工作，保存其状态。等到应用程序终止时进行处理就晚了。

当收到暂停事件时，应用程序应该将其状态存储在磁盘上。如果再次启动应用程序，应用程序可以显示给用户，好像它从未终止。只需要把页面堆栈的信息存储到用户退出的页面上，恢复页面堆栈，并把字段初始化为用户输入的数据，就允许用户返回。

本节的示例应用程序 ApplicationLifetime 就完成这个任务。在这个程序中，允许在多个页面之间的导航，可以输入状态。应用程序暂停时，存储页面堆栈和状态，在启动应用程序时恢复它们。

## 36.2.1 应用程序的执行状态

应用程序的状态使用 ApplicationExecutionState 枚举定义。该枚举定义了 NotRunning、Running、Suspended、Terminated 和 ClosedByUser 状态。应用程序需要知道并存储自己的状态，因为用户在返回应用程序时希望继续原来的操作。

在 App 类的 OnLaunched 方法中，可以使用 LauchActivatedEventArgs 参数的 PreviousExecutionState 属性获取应用程序的前一个执行状态。如果应用程序是在安装后第一次启动，在重启计算机后启动，或者用户上一次在任务管理器中终止了其进程，那么该应用程序的前一个状态是 NotRunning。如果用户单击应用程序的图标时应用程序已经激活，或者应用程序通过某个激活协定激活，则其前一个执行状态为 Running。如果应用程序被暂停，那么激活它时 PreviousExecutionState 属性会返回 Suspended。一般来说，在这种情况中不需要执行什么特殊操作。因为状态仍在内存中可用。在暂停状态下，应用程序不使用 CPU 循环，也没有磁盘访问。

**注意：**

*应用程序可以实现一个或多个协定，然后用其中一个协定激活应用程序。这类协定的一个例子是共享。使用这个协定，用户可以共享另一个应用程序中的一些数据，并使用它作为共享目标，启动一个 Windows 应用程序。实现共享协定参见本章的“共享数据”一节。*

## 36.2.2 在页面之间导航

展示 Windows 应用程序生命周期的示例应用程序(ApplicationLifetime)从 Blank App 模板开始。创建项目后，添加页面 Page1 和 Page2，实现页面之间的导航。

在 MainPage 中，添加两个按钮控件来导航 Page2 和 Page1，再添加两个文本框控件，在导航时传递数据(代码文件 ApplicationLifetime/MainPage.xaml)：

```
<Button Content="Page 1" Click="{x:Bind GotoPage1, Mode=OneTime}"
  Grid.Row="1" />
<TextBox Header="Parameter 1" Text="{x:Bind ParameterPage1, Mode=TwoWay}"
  Grid.Row="1" Grid.Column="1" />
<Button Content="Page 2" Click="{x:Bind GotoPage2, Mode=OneTime}"
  Grid.Row="2" />
<TextBox Header="Parameter 2" Text="{x:Bind Parameter2, Mode=TwoWay}"
  Grid.Row="2" Grid.Column="1" />
```

代码隐藏文件包含事件处理程序、Page1和Page2的导航代码，以及参数的属性(代码文件ApplicationLifetime/MainPage.xaml.cs)：

```
public void GotoPage1() =>
  Frame.Navigate(typeof(Page1), ParameterPage1);

public string ParameterPage1 { get; set; }

public void GotoPage2() =>
  Frame.Navigate(typeof(Page2), ParameterPage2);

public string ParameterPage2 { get; set; }
```

Page1 的 UI 元素显示在导航到这个页面时接收到的数据、允许用户导航到 Page2 的按钮，以及允许用户输入一些状态信息的一个文本框，应用程序终止时会保存这些状态信息(代码文件 ApplicationLifetime/Page1.xaml)：

```
<TextBlock Text="Page 1" Style="{StaticResource HeaderTextBlockStyle}" />
<TextBlock Grid.Row="1" Text="{x:Bind ReceivedContent, Mode=OneTime}"
  Style="{StaticResource BodyTextBlockStyle}" Margin=12 />
<TextBox Grid.Row="2" Text="{x:Bind Parameter1, Mode=TwoWay}" />
<Button Grid.Row="3" Content="Navigate to Page 2"
  Click="{x:Bind GotoPage2, Mode=OneTime}" />
<TextBox Header="Session State 1" Grid.Row="4"
  Text="{x:Bind Data.Session1, Mode=TwoWay}" />
<TextBox Header="Session State 2" Grid.Row="5"
  Text="{x:Bind Data.Session2, Mode=TwoWay}" />
```

类似于 MainPage，Page1 的导航代码为导航时传递的数据定义了一个自动实现的属性，和实现导航到 Page2 的一个事件处理程序(代码文件 ApplicationLifetime/Page1.xaml.cs)：

```
public void GotoPage2() => Frame.Navigate(typeof(Page2), Parameter1);

public string Parameter1 { get; set; }
```

在代码隐藏文件中，导航参数在 OnNavigatedTo 方法的重写版本中接收。接收到的参数分配给自动实现的属性 ReceivedContent(代码文件 ApplicationLifetimeSample/Page1.xaml.cs)：

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
  base.OnNavigatedTo(e);
  //...
  ReceivedContent = e.Parameter?.ToString() ?? string.Empty;
  Bindings.Update();
}

public string ReceivedContent { get; private set; }
```

在导航的实现代码中，Page2 非常类似于 Page1，所以这里不重复它的实现。

使用第 33 章介绍的系统后退按钮，这里，后退按钮的可见性和处理程序在类 BackButtonManager 中定义。如果框架实例的 CanGoBack 属性返回 true，那么构造函数的实现代码使后退按钮可见。如果堆栈可用，就实现 OnBackRequested 方法，返回页面堆栈(代码文件 ApplicationLifetime/Utilities/BackButtonManager.cs)：

```
public class BackButtonManager: IDisposable
{
  private SystemNavigationManager _navigationManager;
  private Frame _frame;

  public BackButtonManager(Frame frame)
  {
    _frame = frame ?? throw new ArgumentNullException(nameof(frame));
    _navigationManager = SystemNavigationManager.GetForCurrentView();
    _navigationManager.AppViewBackButtonVisibility = frame.CanGoBack ?
      AppViewBackButtonVisibility.Visible:
```

```
            AppViewBackButtonVisibility.Collapsed;
        _navigationManager.BackRequested += OnBackRequested;
      }

      private void OnBackRequested(object sender, BackRequestedEventArgs e)
      {
        if (_frame.CanGoBack) _frame.GoBack();
        e.Handled = true;
      }

      public void Dispose()
      {
        _navigationManager.BackRequested -= OnBackRequested;
      }
    }
```

在所有页面中，通过在 OnNavigatedTo 方法中传递 Frame 来实例化 BackButtonManager，它在 OnNavigatedFrom 方法中销毁(代码文件 ApplicationLifetime/MainPage.xaml.cs)：

```
private BackButtonManager _backButtonManager;
protected override void OnNavigatedTo(NavigationEventArgs e)
{
  base.OnNavigatedTo(e);
  _backButtonManager = new BackButtonManager(Frame);
}

protected override void OnNavigatingFrom(NavigatingCancelEventArgs e)
{
  base.OnNavigatingFrom(e);
  _backButtonManager.Dispose();
}
```

有了所有这些代码，用户可以在 3 个不同的页面之间后退和前进。下一步需要记住页面和页面堆栈，把应用程序导航到用户最近一次访问的页面上。

## 36.3 导航状态

为了存储和加载导航状态，类 NavigationSuspensionManager 定义了方法 SetNavigationStateAsync 和 GetNavigationStateAsync。导航的页面堆栈可以在单个字符串中表示。这个字符串写入本地缓存文件中，用一个常数给它命名。如果应用程序以前运行时文件已经存在，就覆盖它。不需要记住应用程序多次运行之间的页面导航(代码文件 ApplicationLifetime/Utilities/NavigationSuspensionManager.cs)：

```
public class NavigationSuspensionManager
{
  private const string NavigationStateFile = "NavigationState.txt";
  public async Task SetNavigationStateAsync(string navigationState)
  {
    StorageFile file = await
      ApplicationData.Current.LocalCacheFolder.CreateFileAsync(
        NavigationStateFile, CreationCollisionOption.ReplaceExisting);
    Stream stream = await file.OpenStreamForWriteAsync();
    using (var writer = new StreamWriter(stream))
    {
      await writer.WriteLineAsync(navigationState);
    }
  }

  public async Task<string> GetNavigationStateAsync()
  {
    Stream stream = await
      ApplicationData.Current.LocalCacheFolder.OpenStreamForReadAsync(
        NavigationStateFile);
    using (var reader = new StreamReader(stream))
    {
      return await reader.ReadLineAsync();
    }
  }
}
```

**注意：**

NavigationSuspensionManager 类利用 Windows 运行库 API 和.NET 的 Stream 类读写文件的内容。这两个功能详见第 22 章。

### 36.3.1　暂停应用程序

为了在暂停应用程序时保存状态，在 OnSuspending 事件处理程序中设置 App 类的 Suspending 事件。当应用程序进入暂停模式时触发事件(代码文件 ApplicationLifetime/App.xaml.cs)：

```
public App()
{
  this.InitializeComponent();
  this.Suspending += OnSuspending;
}
```

OnSuspending 是一个事件处理程序方法，因此声明为返回 void。这有一个问题。只要方法完成，应用程序就可以终止。然而，因为方法声明为 void，所以不可能等待方法完成。因此，收到的 SuspendingEventArgs 参数定义了一个 SuspendingDeferral，通过调用 GetDeferral 方法可以检索它。一旦完成代码的异步功能，需要调用 Complete 方法来延迟。这样，调用者知道方法已完成，应用程序可以终止(代码文件 ApplicationLifetime/App.xaml.cs)：

```
private async void OnSuspending(object sender, SuspendingEventArgs e)
{
  var deferral = e.SuspendingOperation.GetDeferral();
  //...
  deferral.Complete();
}
```

**注意：**

异步方法参见第 15 章。

在 OnSuspending 方法的实现中，页面堆栈写入临时缓存。使用 Frame 的 BackStack 属性可以在页面堆栈上检索页面。这个属性返回 PageStackEntry 对象的列表，其中每个实例代表类型、导航参数和导航过渡信息。为了用 SetNavigationStateAsync 方法存储页面跟踪，只需要一个字符串，其中包含完整的页面堆栈信息。这个字符串可以通过调用 Frame 的 GetNavigationState 方法来检索(代码文件 ApplicationLifetime/App.xaml.cs )：

```
private async void OnSuspending(object sender, SuspendingEventArgs e)
{
  var deferral = e.SuspendingOperation.GetDeferral();
  var frame = Window.Current.Content as Frame;
  if (frame?.BackStackDepth >= 1)
  {
    var suspensionManager = new NavigationSuspensionManager();
    string navigationState = frame.GetNavigationState();
    if (navigationState != null)
    {
      await suspensionManager.SetNavigationStateAsync(navigationState);
    }
  }
  //...
  deferral.Complete();
}
```

默认情况下，在应用程序终止前，只暂停几秒钟。但是，可以延长这个时间，以进行网络调用，从服务中检索数据，给服务上传数据，或跟踪位置。为此，只需要在 OnSuspending 方法内创建一个 ExtendedExecutionSession，设置理由，比如 ExtendedExecutionReason.SavingData。调用 RequestExecutionAsync 来请求扩展。只要没有拒绝延长应用程序的执行，就可以继续扩展的任务。

### 36.3.2 激活暂停的应用程序

GetNavigationState 返回的字符串用逗号分隔，列出了页面堆栈的完整信息，包括类型信息和参数。不应该解析字符串，获得其中的不同部分，因为在 Windows 运行库的更新实现中，这可能会改变。仅仅使用这个字符串恢复状态，用 SetNavigationState 恢复页面堆栈是可行的。如果字符串格式在未来的版本中有变化，这两个方法也会改变。

在启动应用程序时，为了设置页面堆栈，需要更改 OnLaunched 方法。这个方法在 Application 基类中重写，在启动应用程序时调用。参数 LaunchActivatedEventArgs 给出了应用程序启动方式的信息。Kind 属性返回一个 ActivationKind 枚举值，通过它可以读取应用程序的启动方式：由用户单击磁贴，启动一个语音命令，或在 Windows 中启动，例如把它启动为一个共享目标。这个场景需要 PreviousExecutionState，它返回一个 ApplicationExecutionState 枚举值，来提供之前应用程序结束方式的信息。如果应用程序用 ClosedByUser 值结束，就不需要特殊操作，应用程序应重新开始。然而，如果应用程序之前是被终止的，PreviousExecutionState 就包含 Terminated 值。这个状态可用于将应用程序返回到之前用户退出时的状态。这里，页面堆栈从 NavigationSuspensionManager 中检索，给方法 SetNavigationState 传递以前保存的字符串，来设置根框架(代码文件 ApplicationLifetime/App.xaml.cs)：

```
protected override async void OnLaunched(LaunchActivatedEventArgs e)
{
  Frame rootFrame = Window.Current.Content as Frame;
  if (rootFrame == null)
  {
    rootFrame = new Frame();
    rootFrame.NavigationFailed += OnNavigationFailed;
    if (e.PreviousExecutionState == ApplicationExecutionState.Terminated)
    {
      var suspensionManager = new NavigationSuspensionManager();
      string navigationState =
        await suspensionManager.GetNavigationStateAsync();
      rootFrame.SetNavigationState(navigationState);
      //...
    }

    // Place the frame in the current Window
    Window.Current.Content = rootFrame;
  }
  if (rootFrame.Content == null)
  {
    rootFrame.Navigate(typeof(MainPage), e.Arguments);
  }
  Window.Current.Activate();
}
```

### 36.3.3 测试暂停

现在启动该应用程序(参见图 36-1)，导航到另一个页面，然后打开另一个应用程序，并等待前一个应用程序终止。如果将 Status Values 选项设置为“Show Suspended Status”，则可以在任务管理器的 Details 视图中看到暂停的应用程序。但是，在测试暂停时，这不是一个简单的方法(因为应用程序可能在很久之后才暂停)，但可以调试不同的状态。

使用调试器则不同。如果应用程序一旦失去焦点就会暂停，那么每到达一个断点就会暂停，因此在调试器中运行时，暂停是被禁用的，正常的暂停机制不会起作用。但是，模拟暂停很容易。打开 Debug Location 工具栏，可以看到 3 个按钮：Suspend、Resume 和 Suspend and shutdown(参见图 36-2)。如果选择 Suspend and shutdown，然后再次启动应用程序，那么应用程序将从前一个状态 ApplicationExecutionState.Terminated 继续运行，因此会打开用户之前打开的页面。

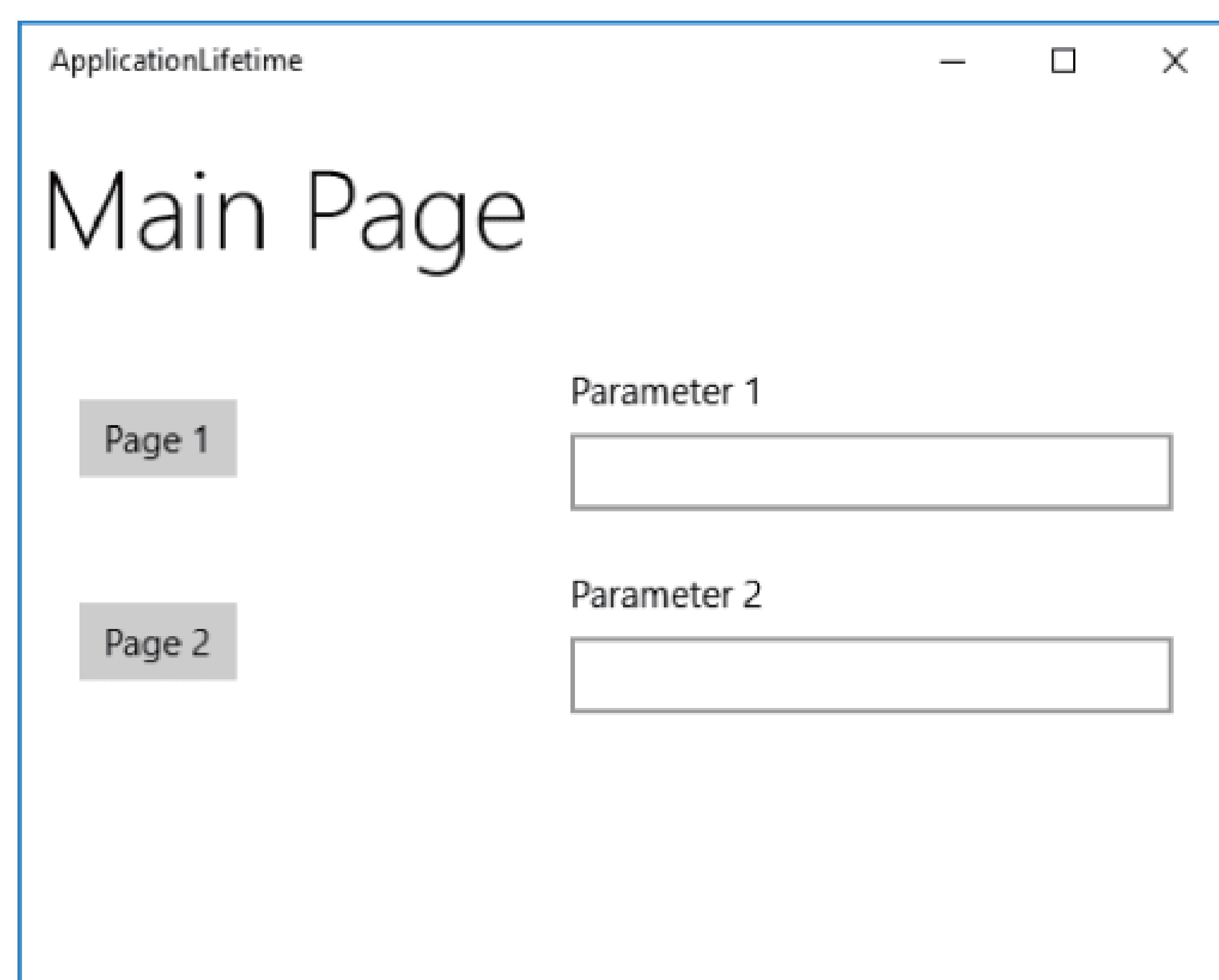



图 36-1

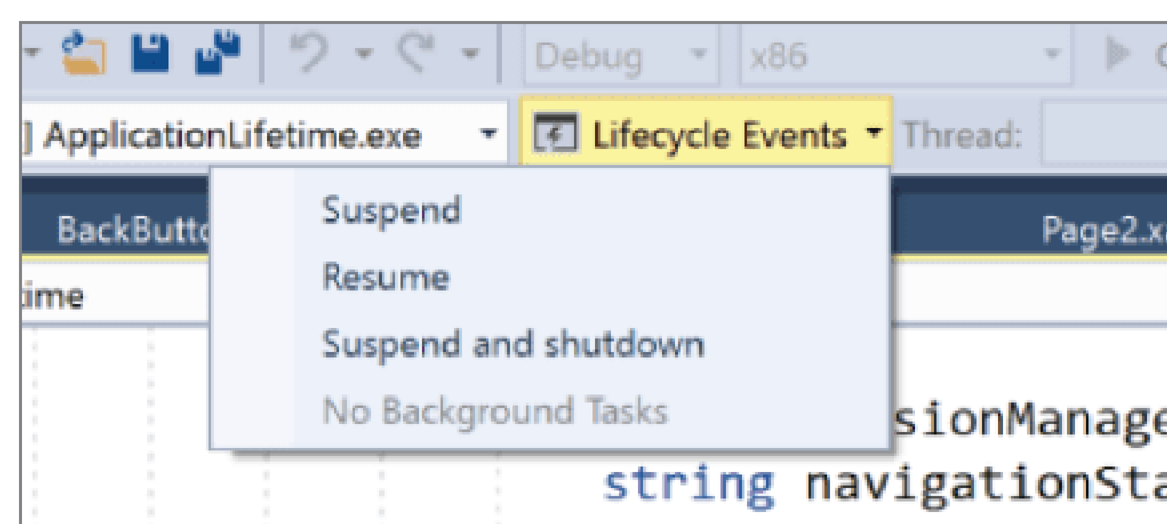



图 36-2

### 36.3.4　页面状态

用户输入的任何数据也应该恢复。为了进行演示，在 Page1 上创建两个输入字段(代码文件 ApplicationLifetime/Page1.xaml)：

```
<TextBox Header="Session State 1" Grid.Row="4"
  Text="{x:Bind Data.Session1, Mode=TwoWay}" />
<TextBox Header="Session State 2" Grid.Row="5"
  Text="{x:Bind Data.Session2, Mode=TwoWay}" />
```

这个输入字段的数据表示由 DataManager 类定义，从 Data 属性中返回，如下面的代码片段所示(代码文件 ApplicationLifetime/Page1.xaml.cs)：

```
public DataManager Data => DataManager.Instance;
```

DataManager 类定义了属性 Session1 和 Session2，其值存储在 Dictionary 中(代码文件 ApplicationLifetime/Services/DataManager.cs)：

```
public class DataManager: INotifyPropertyChanged
{
  private const string SessionStateFile = "TempSessionState.json";
  private Dictionary<string, string> _state = new Dictionary<string, string>()
  {
    [nameof(Session1)] = string.Empty,
    [nameof(Session2)] = string.Empty
  };

  private DataManager()
  {
  }

  public event PropertyChangedEventHandler PropertyChanged;
  protected void OnPropertyChanged(
    [CallerMemberName] string propertyName = null) =>
    PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));

  public static DataManager Instance { get; } = new DataManager();

  public string Session1
  {
    get => _state[nameof(Session1)];
    set
    {
      _state[nameof(Session1)] = value;
      OnPropertyChanged();
    }
  }

  public string Session2
  {
```

```
    get => _state[nameof(Session2)];
    set
    {
      _state[nameof(Session2)] = value;
      OnPropertyChanged();
    }
  }
  //...
}
```

为了加载和存储会话状态，定义了 SaveTempSessionAsync 和 LoadTempSessionAsync 方法。其实现代码使用 Json.Net 将字典序列化为 JSON 格式。但是，可以使用任何序列化(代码文件 ApplicationLifetime/Services/DataManager.cs):

```
public async Task SaveTempSessionAsync()
{
  StorageFile file =
    await ApplicationData.Current.LocalCacheFolder.CreateFileAsync(
      SessionStateFile, CreationCollisionOption.ReplaceExisting);
  Stream stream = await file.OpenStreamForWriteAsync();
  var serializer = new JsonSerializer();
  using (var writer = new StreamWriter(stream))
  {
    serializer.Serialize(writer, _state);
  }
}

public async Task LoadTempSessionAsync()
{
  Stream stream = await
    ApplicationData.Current.LocalCacheFolder.OpenStreamForReadAsync(
      SessionStateFile);
  var serializer = new JsonSerializer();
  using (var reader = new StreamReader(stream))
  {
    string json = await reader.ReadLineAsync();
    Dictionary<string, string> state =
      JsonConvert.DeserializeObject<Dictionary<string, string>>(json);
    _state = state;
    foreach (var item in state)
    {
      OnPropertyChanged(item.Key);
    }
  }
}
```

**注意：**

XML 和 JSON 的序列化参见网上附加第 2 章。

剩下的就是调用 SaveTempSessionAsync 和 LoadTempSessionAsync 方法，暂停、激活应用程序。这些方法添加到 OnSuspending 和 OnLaunched 方法中读写页面堆栈的地方(代码文件 Application Lifetime/App.xaml.cs)：

```
private async void OnSuspending(object sender, SuspendingEventArgs e)
{
  var deferral = e.SuspendingOperation.GetDeferral();
  //...
  await DataManager.Instance.SaveTempSessionAsync();
  deferral.Complete();
}

protected override async void OnLaunched(LaunchActivatedEventArgs e)
{
  Frame rootFrame = Window.Current.Content as Frame;
  if (rootFrame == null)
  {
    rootFrame = new Frame();
    rootFrame.NavigationFailed += OnNavigationFailed;
    if (e.PreviousExecutionState == ApplicationExecutionState.Terminated)
    {
      //...
      await DataManager.Instance.LoadTempSessionAsync();
    }
    // Place the frame in the current Window
```

```
    Window.Current.Content = rootFrame;
  }
  if (rootFrame.Content == null)
  {
    rootFrame.Navigate(typeof(MainPage), e.Arguments);
  }
  Window.Current.Activate();
}
```

现在，可以运行应用程序，在 Page2 中输入状态，暂停和终止程序，再次启动它，再次显示状态。

在应用程序的生命周期中，需要为 Windows 应用程序进行特殊的编程，以考虑电池的耗费。下一节讨论在应用程序间共享数据，这也可以用于手机平台。

## 36.4　共享数据

如果应用程序提供与其他应用程序的交互，就会更有用。在 Windows 10 中，应用程序可以使用拖放操作共享数据，甚至桌面应用程序也这样做。在 Windows 应用程序之间，也可以使用共享协定分享数据。

使用共享协定时，一个应用程序(共享源)可以用许多不同的格式共享数据，例如文本、HTML、图片或自定义数据，用户可以选择接收数据格式的应用程序，作为共享目标。Windows 使用安装时应用程序注册的协定，找到支持相应数据格式的应用程序。

### 36.4.1　共享源

关于共享，首先要考虑的是确定哪些数据以何种格式共享。可以共享简单文本、富文本、HTML 和图像，也可以共享自定义类型。当然，其他应用程序(即共享目标)必须知道且能使用所有这些类型。对于自定义类型，只有知道该类型且是该类型的共享目标的应用程序才能共享它。示例应用程序提供了文本格式的数据和 HTML 格式的图书列表。

为了用 HTML 格式提供图书信息，定义了一个简单的 Book 类(代码文件 SharingSource/Models/Book.cs)：

```
public class Book
{
  public string Title { get; set; }
  public string Publisher { get; set; }
}
```

Book 对象列表从 BooksRepository 类的 GetSampleBooks 方法中返回(代码文件 SharingDate/SharingSource/Models/BooksRepository.cs)：

```
public class BooksRepository
{
  public IEnumerable<Book> GetSampleBooks() =>
    new List<Book>()
    {
      new Book
      {
        Title = "Professional C# 7 and .NET Core 2",
       Publisher = "Wrox Press"
      },
      new Book
      {
        Title = "Professional C# 6 and .NET Core 1.0",
        Publisher = "Wrox Press"
      }
    };
}
```

要把 Book 对象列表转换为 HTML，扩展 ToHtml 方法通过 LINQ to XML 返回一个 HTML 表(代码文件 SharingData/SharingSource/Utilities/BooksExtensions.cs)：

```
public static class BookExtensions
{
  public static string ToHtml(this IEnumerable<Book> books) =>
    new XElement("table",
      new XElement("thead",
```

```
        new XElement("tr",
          new XElement("td", "Title"),
          new XElement("td", "Publisher"))),
      books.Select(b =>
        new XElement("tr",
          new XElement("td", b.Title),
          new XElement("td", b.Publisher)))).ToString();
}
```

**注意：**

LINQ to XML 参见网上附加第 2 章。

在 MainPage 中定义了一个按钮，用户可以通过它启动共享，再定义一个文本框控件，供用户输入要共享的文本数据 (代码文件 SharingData/Share Source/MainPage.xaml)：

```
<RelativePanel Margin="24">
  <Button x:Name="shareDataButton" Content="Share Data"
    Click="{x:Bind DataSharing.ShowShareUI, Mode=OneTime}" Margin="12" />
  <TextBox RelativePanel.RightOf="shareDataButton"
    Text="{x:Bind DataSharing.SimpleText, Mode=TwoWay}" Margin="12" />
</RelativePanel>
```

在代码隐藏文件中，DataSharing 属性返回 ShareDataViewModel，其中实现了所有重要的分享功能(代码文件 SharingData/Share Source/MainPage.xaml.cs)：

```
public ShareDataViewModel DataSharing { get; set; } = new ShareDataViewModel();
```

ShareDataViewModel 定义了 XAML 文件绑定的属性 SimpleText，用于输入要共享的简单文本。对于分享，把事件处理程序方法 ShareDataRequested 分配给 DataTransferManager 的事件 DataRequested。用户请求共享数据时，触发这个事件(代码文件 SharingData/Share Source/ViewModels/ShareDataViewModel.cs)：

```
public class ShareDataViewModel
{
  public ShareDataViewModel()
  {
    DataTransferManager.GetForCurrentView().DataRequested +=
      ShareDataRequested;
  }

  public string SimpleText { get; set; } = string.Empty;
  //...
}
```

当触发事件时，调用 OnShareDataRequested 方法。这个方法接收 DataTransferManager 作为第一个参数，DataRequestedEventArgs 作为第二个参数。在共享数据时，需要填充 args.Request.Data 引用的 DataPackage。可以使用 Title、Description 和 Thumbnail 属性给用户界面提供信息。应共享的数据必须用一个 SetXXX 方法传递。示例代码分享一个简单的文本和 HTML 代码，因此使用方法 SetText 和 SetHtmlFormat。HtmlFormatHelper 类帮助创建需要共享的 HTML 代码。图书的 HTML 代码用前面的扩展方法 ToHtml 创建(代码文件 SharingData/Share Source/ViewModels/ShareDataViewModel.cs)：

```
private void ShareDataRequested(DataTransferManager sender,
  DataRequestedEventArgs args)
{
  var books = new BooksRepository().GetSampleBooks();
  Uri baseUri = new Uri("ms-appx:///");
  DataPackage package = args.Request.Data;
  package.Properties.Title = "Sharing Sample";
  package.Properties.Description = "Sample for sharing data";
  package.Properties.Thumbnail = RandomAccessStreamReference.CreateFromUri(
    new Uri(baseUri, "Assets/Square44x44Logo.png"));
  package.SetText(SimpleText);
  package.SetHtmlFormat(HtmlFormatHelper.CreateHtmlFormat(books.ToHtml()));
}
```

如果需要共享操作何时完成的信息，例如从源应用程序中删除数据，DataPackage 类就触发 OperationCompleted 和 Destroyed 事件。

**注意：**
除了提供文本或 HTML 代码之外，其他方法，比如 SetBitmap、SetRtf 和 SetUri，也可以提供其他数据格式。

**注意：**
如果需要在 ShareDataRequested 方法中使用异步方法构建要共享的数据，需要使用一个延期，在数据可用时提供信息。这类似于本章前面介绍的页面暂停机制。使用 DataRequestedEventArgs 类型的 Request 属性，可以调用 GetDeferral 方法。这个方法返回一个 DataRequestedDeferral 类型的延期。使用这个对象，可以在数据可用时调用 Complete 方法。

最后，需要显示分享的用户界面。这允许用户选择目标应用程序：

```
public void ShowShareUI()
{
  DataTransferManager.ShowShareUI();
}
```

图 36-3 展示了调用 DataTransferManager 的 ShowShareUI 方法后的用户界面。根据所提供的数据格式和安装的应用程序，显示相应的应用程序，作为选项。从 Windows 10 的 Fall Creators Update (构建号 16299)版本开始，不仅可以看到应用程序，还可以看到一些联系人。选择联系人时，可以使用应用程序与联系人通信，直接与联系人共享数据。

如果选择 Mail 应用，就传递 HTML 信息。图 36-4 显示在这个应用程序中接收的数据。

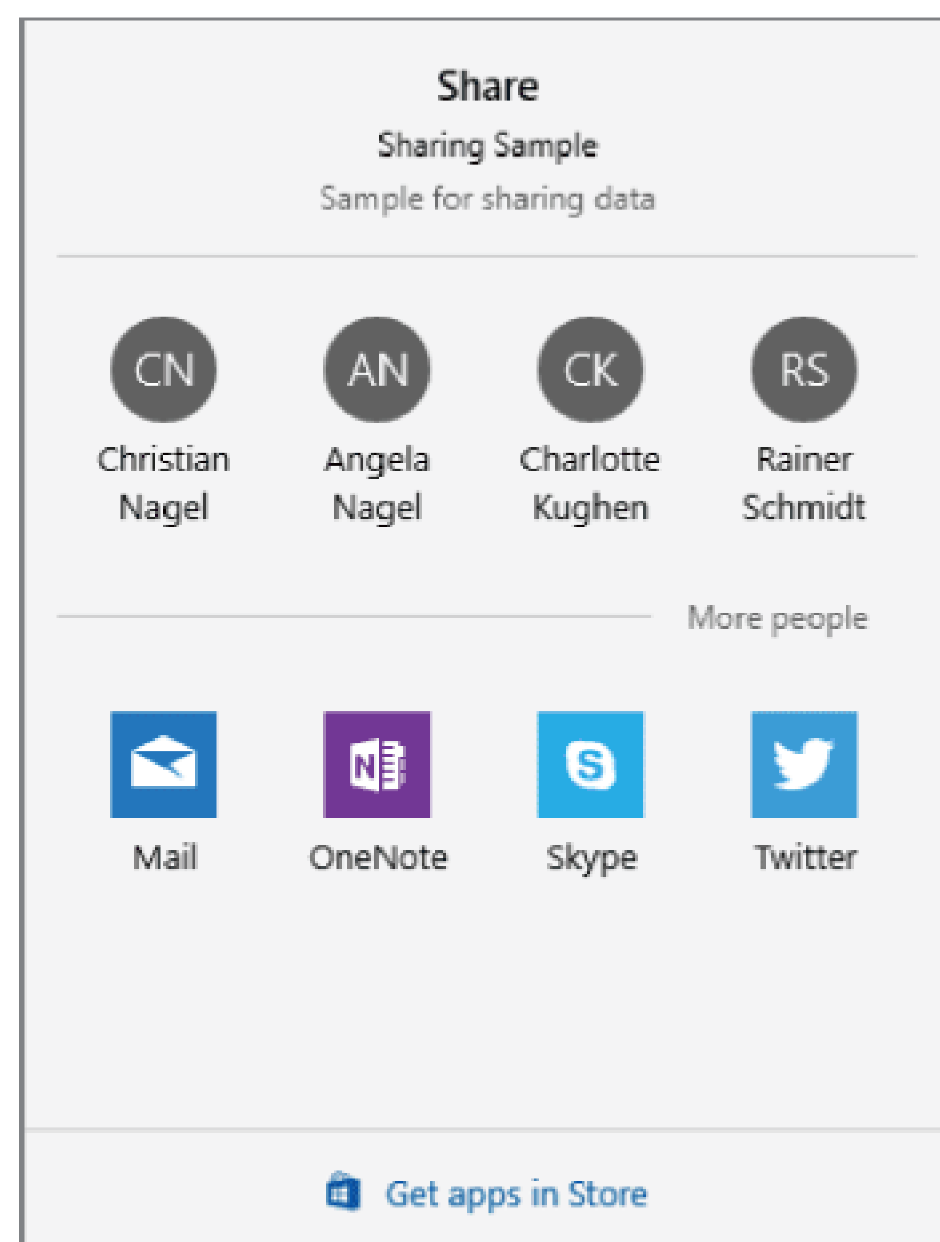



图 36-3

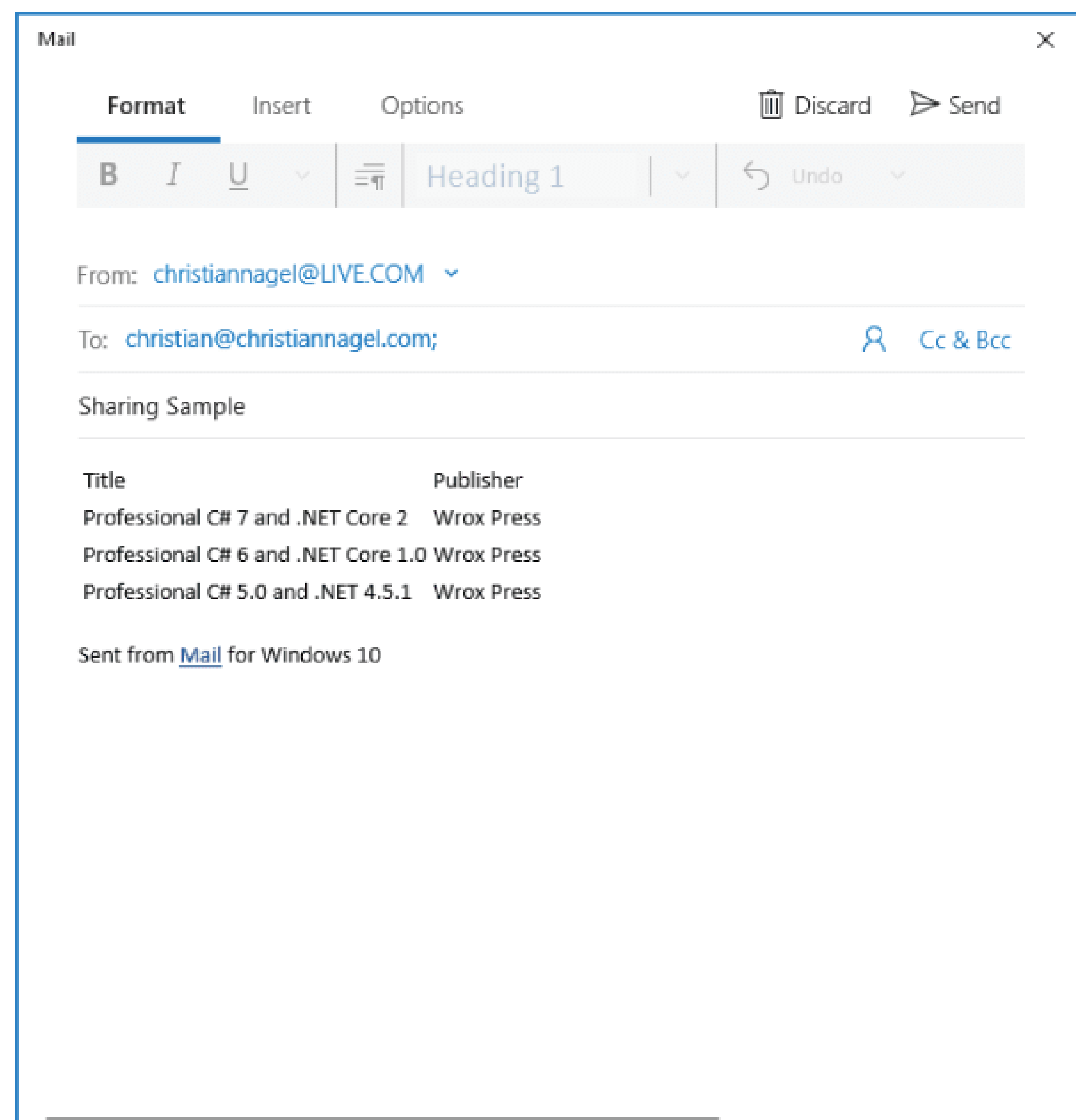



图 36-4

**注意：**
共享数据最初是在 Windows 8 中使用功能区设计的。用户必须从右边滑动来访问共享特性。许多用户无法找到这个功能。对于 Windows 10，需要显式地提供一个可见控件(例如，按钮)，用户可以在其中开始共享。

### 36.4.2 共享目标

现在看看共享内容的接收者。如果应用程序应从共享源中接收信息，就需要将其声明为共享目标。图 36-5 显示了清单设计器在 Visual Studio 中的 Declarations 页面，在其中可以定义共享目标。在这里添加 Share Target 声明，它至少要包含一种数据格式。可能的数据格式是 Text、URI、Bitmap、HTML、StorageItems 或 RTF。还可以添加文件扩展名，以指定应支持哪些文件类型。

在注册应用程序时，要使用软件包清单中的信息。这告诉 Windows，哪些应用程序可用作共享目标。示例应用程序 SharingTarget 为 Text 和 HTML 定义了共享目标。

用户把应用程序启动为共享目标时，就在 App 类中调用 OnShareTargetActivated 方法，而不是 OnLaunched 方法。这里创建另一个页面(ShareTargetPage)，显示用户选择这款应用程序作为共享目标时的屏幕(代码文件 SharingData/ShareTarget/App.xaml.cs)：

```
protected override void OnShareTargetActivated(
  ShareTargetActivatedEventArgs args)
{
  Frame rootFrame = CreateRootFrame();
  rootFrame.Navigate(typeof(ShareTargetPage), args.ShareOperation);
  Window.Current.Activate();
}
```

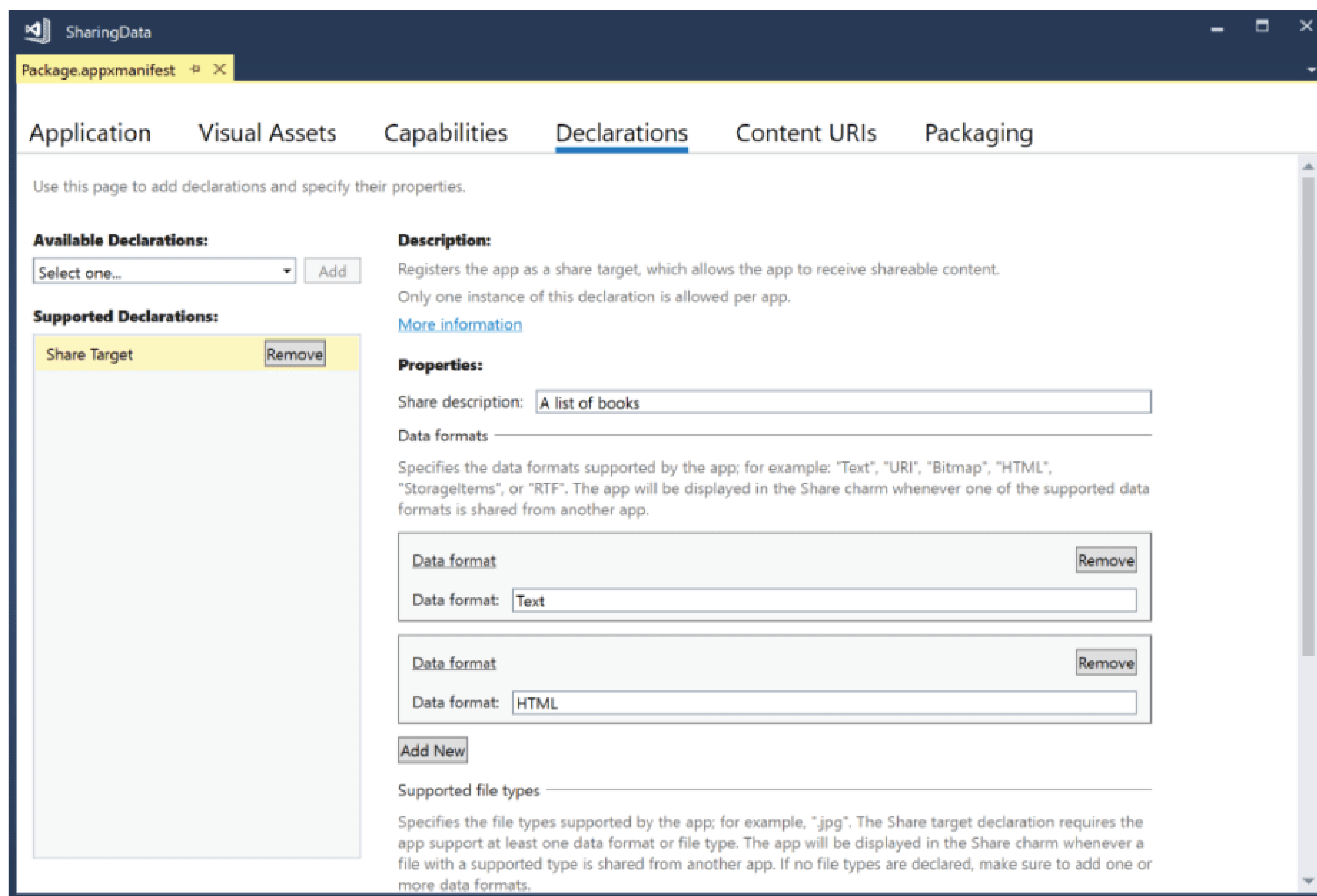



图 36-5

为了不在两个不同的地方创建根框架，应该重构 OnLaunched 方法，把框架创建代码放在一个单独的方法 CreateRootFrame 中。这个方法在 OnShareTargetActivated 和 OnLaunched 中调用：

```
private Frame CreateRootFrame()
{
  Frame rootFrame = Window.Current.Content as Frame;
  if (rootFrame == null)
  {
    rootFrame = new Frame();
    rootFrame.NavigationFailed += OnNavigationFailed;
    Window.Current.Content = rootFrame;
  }
  return rootFrame;
}
```

OnLaunched 方法的变化如下所示。与 OnShareTargetActivated 相反，这个方法导航到 MainPage：

```
protected override void OnLaunched(LaunchActivatedEventArgs e)
{
  Frame rootFrame = CreateRootFrame();
  if (!e.PreLaunchActivated)
  {
    if (rootFrame.Content == null)
    {
      rootFrame.Navigate(typeof(MainPage), e.Arguments);
    }
    Window.Current.Activate();
  }
}
```

ShareTargetPage 包含控件，用户可以在其中看到共享数据的信息，如标题和描述，还包括一个组合框，显示了用户可以选择的可用数据格式(代码文件 SharingData/SharingTarget/ShareTargetPage.xaml)：

```
<StackPanel Orientation="Vertical">
  <TextBlock Text="Share Target Page" />
  <TextBox Header="Title" IsReadOnly="True"
    Text="{x:Bind ViewModel.Title, Mode=OneWay}" Margin="12" />
  <TextBox Header="Description" IsReadOnly="True"
    Text="{x:Bind ViewModel.Description, Mode=OneWay}" Margin="12" />
  <ComboBox ItemsSource="{x:Bind ViewModel.ShareFormats, Mode=OneTime}"
    SelectedItem="{x:Bind ViewModel.SelectedFormat, Mode=TwoWay}"
    Margin="12" />
  <Button Content="Retrieve Data"
    Click="{x:Bind ViewModel.RetrieveData, Mode=OneTime}" Margin="12" />
  <Button Content="Report Complete"
    Click="{x:Bind ViewModel.ReportCompleted, Mode=OneTime}" Margin="12" />
  <TextBox Header="Text" IsReadOnly="True"
    Text="{x:Bind ViewModel.Text, Mode=OneWay}" Margin="12" />
  <TextBox AcceptsReturn="True" IsReadOnly="True"
    Text="{x:Bind ViewModel.Html, Mode=OneWay}" Margin="12" />
</StackPanel>
```

在代码隐藏文件中，把 ShareTargetPageViewModel 分配给 ViewModel 属性。在前面的 XAML 代码中，这个属性使用了编译绑定。另外在 OnNavigatedTo 方法中，把 ShareOperation 对象传递给 Activate 方法， 激活 SharedTargetPageViewModel (代码文件 SharingTarget/ShareTargetPage.xaml.cs)：

```
public sealed partial class ShareTargetPage: Page
{
  public ShareTargetPage() => InitializeComponent();

  public ShareTargetPageViewModel ViewModel { get; } =
    new ShareTargetPageViewModel();

  protected override void OnNavigatedTo(NavigationEventArgs e)
  {
    ViewModel.Activate(e.Parameter as ShareOperation);
    base.OnNavigatedTo(e);
  }
}
```

类 ShareTargetPageViewModel 为应该显示在页面中的值定义了属性，还实现 INotifyProperty Changed 接口，为更改通知定义了属性(代码文件 SharingTarget/ViewModels/ShareTargetViewModel.cs)：

```
public class ShareTargetPageViewModel: INotifyPropertyChanged
{
  public event PropertyChangedEventHandler PropertyChanged;

  public void OnPropertyChanged(
    [CallerMemberName] string propertyName = null) =>
    PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));

  public void Set<T>(ref T item, T value,
    [CallerMemberName] string propertyName = null)
  {
    if (!EqualityComparer<T>.Default.Equals(item, value))
    {
      item = value;
      OnPropertyChanged(propertyName);
    }
  }
```

```
    //...

    private string _text;
    public string Text
    {
      get => _text;
      set => Set(ref _text, value);
    }

    private string _html;
    public string Html
    {
      get => _html;
      set => Set(ref _html, value);
    }

    private string _title;
    public string Title
    {
      get => _title;
      set => Set(ref _title, value);
    }

    private string _description;
    public string Description
    {
      get => _description;
      set => Set(ref _description, value);
    }
  }
```

Activate 方法是 ShareTargetPageViewModel 的一个重要部分。这里，ShareOperation 对象用于访问共享数据的信息，得到一些可用于显示给用户的元数据，如 Title、Description 和可用数据格式的列表。如果出错，就调用 ShareOperation 的 ReportError 方法，把错误信息显示给用户(代码文件 SharingTarget/ViewModels/ShareTargetViewModel.cs)：

```
  public class ShareTargetPageViewModel: INotifyPropertyChanged
  {
    //...
    private ShareOperation _shareOperation;
    private readonly ObservableCollection<string> _shareFormats =
      new ObservableCollection<string>();
    public string SelectedFormat { get; set; }
    public IEnumerable<string> ShareFormats => _shareFormats;

    public void Activate(ShareOperation shareOperation)
    {
      if (shareOperation == null)
        throw new ArgumentNullException(nameof(shareOperation));

      string title = null;
      string description = null;
      try
      {
        _shareOperation = shareOperation;
        title = _shareOperation.Data.Properties.Title;
        description = _shareOperation.Data.Properties.Description;
        foreach (var format in _shareOperation.Data.AvailableFormats)
        {
          _shareFormats.Add(format);
        }
        Title = title;
        Description = description;
      }
      catch (Exception ex)
      {
        _shareOperation.ReportError(ex.Message);
      }
    }
    //...
  }
```

一旦用户选择数据格式，可以单击按钮，检索数据。这会调用 RetrieveData 方法。根据用户的选择，在 Data 属性返回的 DataPackageView 实例上调用 GetTextAsync 或 GetHtmlFormatAsync。在检索数据前，调用方法 ReportStarted；检索到数据后，调用方法 ReportDataRetrieved(代码文件 SharingTarget/ViewModels/ShareTargetViewModel.cs)：

```
public class ShareTargetPageViewModel: INotifyPropertyChanged
{
  //...
  private bool dataRetrieved = false;
  public async void RetrieveData()
  {
    try
    {
      if (dataRetrieved)
      {
        await new MessageDialog("data already retrieved").ShowAsync();
      }
      _shareOperation.ReportStarted();
      switch (SelectedFormat)
      {
        case "Text":
          Text = await _shareOperation.Data.GetTextAsync();
          break;
        case "HTML Format":
          Html = await _shareOperation.Data.GetHtmlFormatAsync();
          break;
        default:
          break;
      }
      _shareOperation.ReportDataRetrieved();
      dataRetrieved = true;
    }
    catch (Exception ex)
    {
      _shareOperation.ReportError(ex.Message);
    }
  }
  //...
}
```

在示例应用程序中，检索到的数据显示在用户界面中。在真正的应用程序中，可以使用任何形式的数据，例如，把它本地存储在客户端上，或者调用自己的 Web 服务并给它传递数据。

最后，用户可以在 UI 中单击 Report Completed 按钮。通过 Click 处理程序，会在视图模型中调用 ReportCompleted 方法，进而在 ShareOperation 实例上调用 ReportCompleted 方法。这个方法关闭对话框(代码文件 SharingTarget/ViewModels/ShareTargetViewModel.cs)：

```
public class ShareTargetPageViewModel: INotifyPropertyChanged
{
  //...
  public void ReportCompleted()
  {
    _shareOperation.ReportCompleted();
  }
  //...
}
```

在应用程序中，可以在检索数据之后调用前面的 ReportCompleted 方法。只要记住，应用程序的对话框关闭时，调用此方法。

要激活 ShareTarget 应用程序，需要运行 ShareTarget 应用程序一次，将其注册为一个共享源。然后，再次启动 ShareSource 应用程序。现在，共享数据时，ShareTarget 被列为一个可供选择的应用程序。选择此应用程序，就会看到正在运行的 ShareTarget 应用程序，如图 36-6 所示。

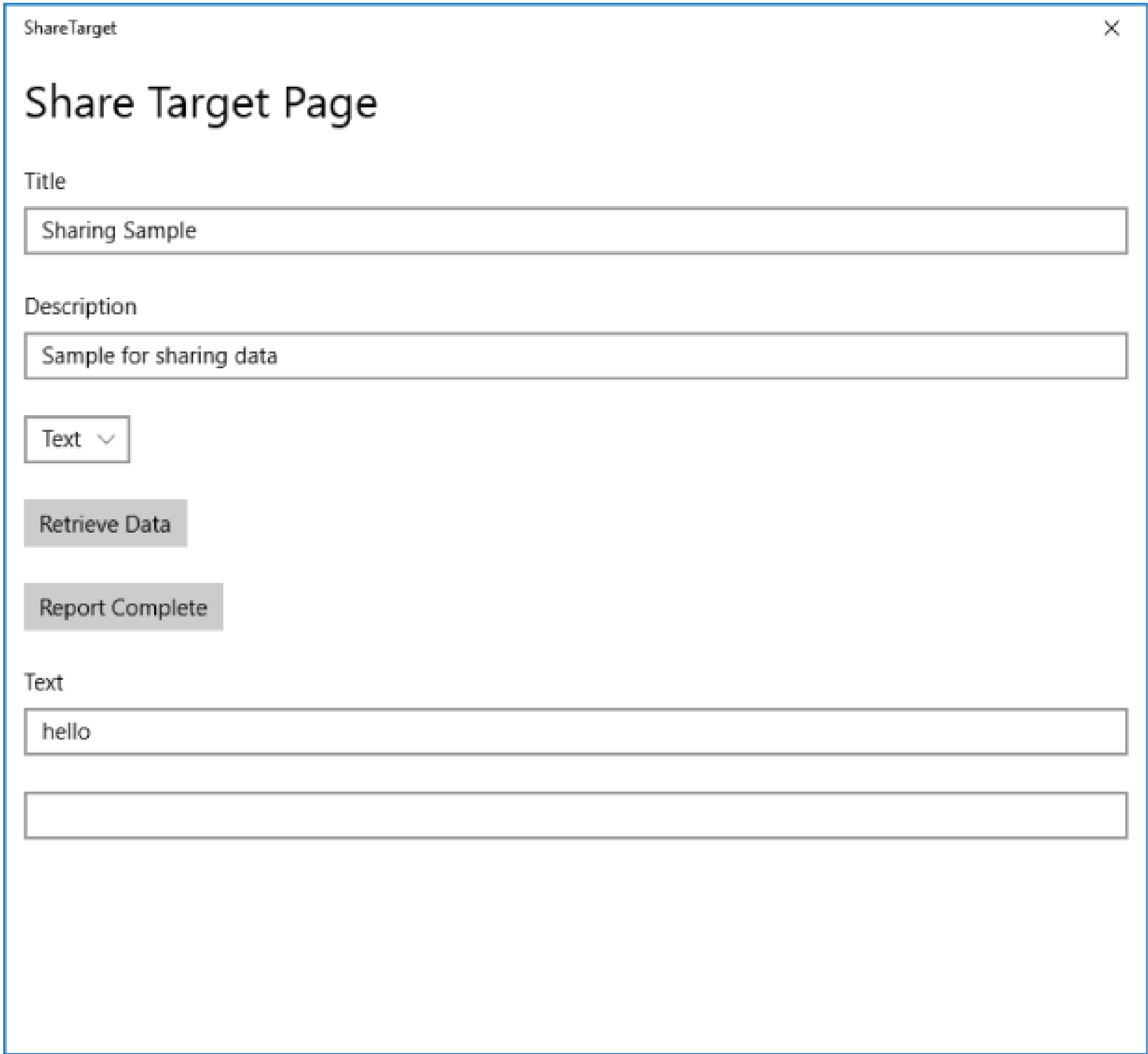



图 36-6

**注意：**

测试分享所有支持格式的最佳方法是使用示例应用程序的 Sharing Content Source 示例和 Sharing Content Target 示例。两个示例应用程序都在 https://github.com/Microsoft/ Windows-universal-samples 上。如果把其中一个应用程序作为共享源，就使用另一个示例应用程序作为目标，反之亦然。

**注意：**

调试共享目标的一个简单方法是把 Debug 选项设置为 Do not launch, but debug my code when it starts。这个设置在 Project Properties 的 Debug 选项卡(参见图 36-7)中。使用此设置，可以启动调试器，一旦与这款应用程序共享数据源应用程序中的数据，应用程序就启动。将共享和目标应用程序都设置为使用调试器启动，目标应用程序在作为共享目标激活后立即启动。

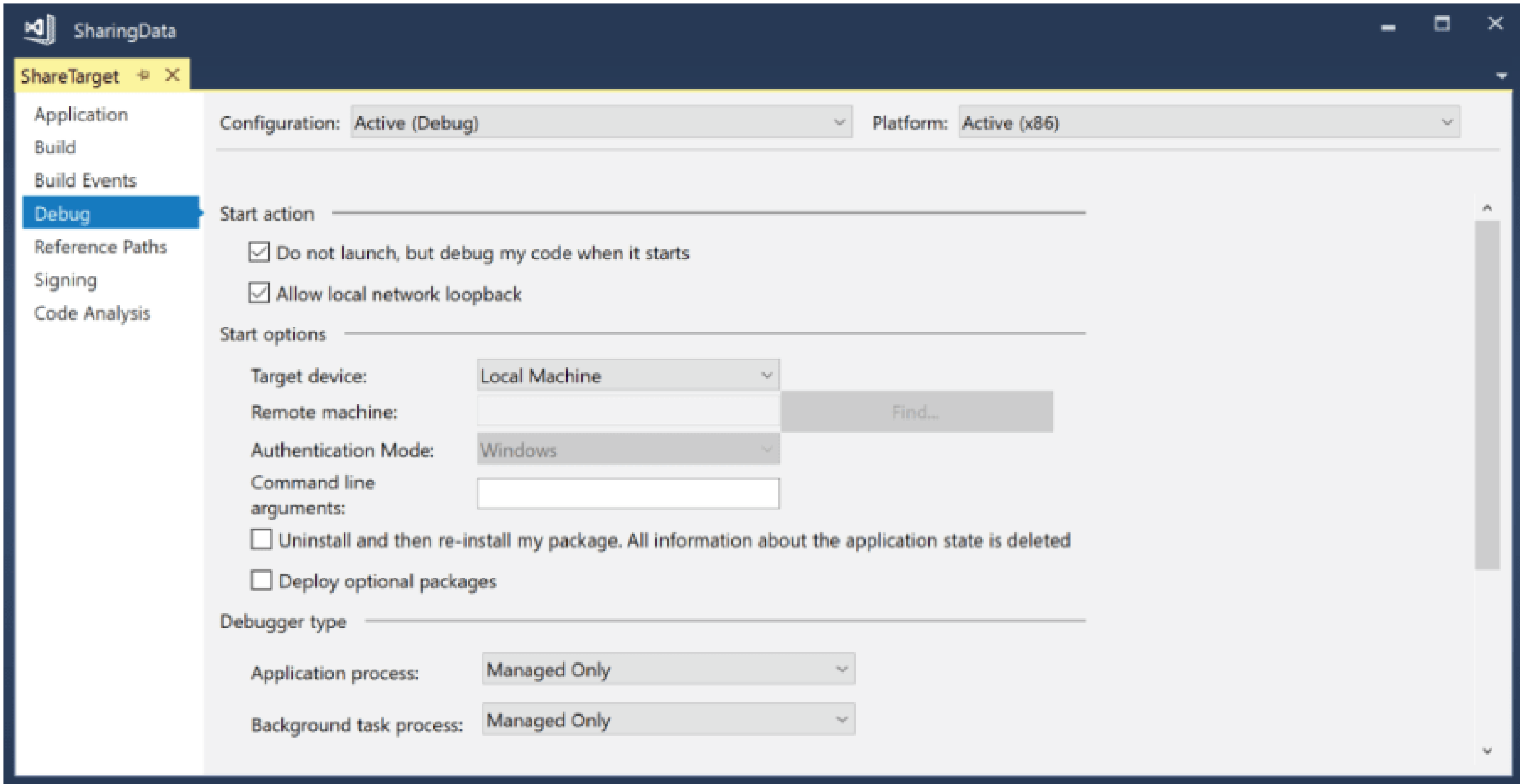



图 36-7

# 36.5　应用程序服务

在应用程序之间共享数据的另一种方法是使用应用程序服务。应用程序服务可以与调用 Web 服务相媲美，但对用户的系统而言，服务是本地的。多个应用程序可以访问相同的服务，这是在应用程序之间共享信息的方式。应用服务和 Web 服务之间的一个重要区别是，用户不需要使用这个特性进行交互，而可以在应用程序中完成。

样例应用程序 AppServices 使用服务缓存 Book 对象。调用服务，可以检索 Book 对象的列表，把新 Book 对象添加到服务中。

应用程序包含多个项目：

- 一个.NET 标准库(BooksCacheModel)定义了这个应用程序的模型：Book 类。为了便于传输数据，提供一个扩展方法，把 Book 对象转换为 JSON，把 JSON 转换为 Book 对象。这个库在所有其他项目中使用。
- 第二个项目(BooksCacheService)是一个 Windows 运行库组件，定义了 book 服务本身。这种服务需要在后台运行，因此实现一个后台任务。
- 后台任务需要注册到系统中。这个项目是一个 Windows 应用程序 BooksCacheProvider。
- 调用应用程序服务的客户机应用程序是一个 Windows 应用程序 BooksCacheClient。

下面看看这些部分。

## 36.5.1　创建模型

移动库 BooksCacheModel 包含 Book 类、利用 NuGet 包 Newtonsoft.Json 转换到 JSON 的转换器以及存储库。

Book 类定义了 Title 和 Publisher 属性(代码文件 AppServices/BooksCacheModel/Book.cs)：

```
public class Book
{
  public string Title { get; set; }
  public string Publisher { get; set; }
}
```

BooksRepository 类包含 Book 对象的内存缓存，允许用户通过 AddBook 方法添加 book 对象，使用 Books 属性返回所有缓存的书。为了查看一本书，不需要添加新书，初始化时把一本书添加到列表中(代码文件 AppServices /BooksCacheModel/BooksRepository.cs)：

```
public class BooksRepository
{
  private readonly List<Book> _books = new List<Book>()
  {
    new Book
    {
      Title = "Professional C# 7 and .NET Core 2",
      Publisher = "Wrox Press"
    }
  };

  public IEnumerable<Book> Books => _books;

  private BooksRepository() { }

  public static BooksRepository Instance = new BooksRepository();

  public void AddBook(Book book) => _books.Add(book);

}
```

因为通过应用程序服务发送的数据需要序列化，所以扩展类 BookExtensions 定义了一些扩展方法，把 Book 对象和 Book 对象列表转换为 JSON 字符串，反之亦然。给应用程序服务传递一个字符串是很简单的。扩展方法利用了 NuGet 包 Newtonsoft.Json 中可用的类 JsonConvert (代码文件 AppServices/BooksCacheModel (BookExtensions.cs)：

```
public static class BookExtensions
{
  public static string ToJson(this Book book) =>
```

```
    JsonConvert.SerializeObject(book);

  public static string ToJson(this IEnumerable<Book> books) =>
    JsonConvert.SerializeObject(books);

  public static Book ToBook(this string json) =>
    JsonConvert.DeserializeObject<Book>(json);

  public static IEnumerable<Book> ToBooks(this string json) =>
    JsonConvert.DeserializeObject<IEnumerable<Book>>(json);
}
```

### 36.5.2 为应用程序服务连接创建后台任务

现在进入这个示例应用程序的核心：应用程序服务。需要把应用服务实现为Windows Runtime组件库，通过实现接口IBackgroundTask把它实现为一个后台任务。Windows后台任务可以在后台运行，不需要用户交互。

有不同种类的后台任务可用。后台任务的启动可以基于定时器的间隔、Windows推送通知、位置信息、蓝牙设备连接或其他事件。

类BooksCacheTask是一个应用程序服务的后台任务。接口IBackgroundTask定义了需要实现的Run方法。在实现代码中，定义了请求处理程序，来接收应用程序服务的连接(代码文件AppServices/BooksCacheService/BooksCacheTask.cs)：

```
public sealed class BooksCacheTask: IBackgroundTask
{
  private BackgroundTaskDeferral _taskDeferral;
  public void Run(IBackgroundTaskInstance taskInstance)
  {
    _taskDeferral = taskInstance.GetDeferral();
    taskInstance.Canceled += OnTaskCanceled;

    var trigger = taskInstance.TriggerDetails as AppServiceTriggerDetails;
    AppServiceConnection connection = trigger.AppServiceConnection;
    connection.RequestReceived += OnRequestReceived;
  }

  private void OnTaskCanceled(IBackgroundTaskInstance sender,
    BackgroundTaskCancellationReason reason)
  {
    _taskDeferral?.Complete();
  }
  //...
}
```

在OnRequestReceived处理程序的实现代码中，服务可以读取请求，且需要提供回应。接收到的请求都包含在AppServiceRequestReceivedEventArgs的Request.Message属性中。Message属性返回一个ValueSet对象。ValueSet是一个字典，其中包含键及其相应的值。这里的服务需要一个command键，其值是GET或POST。GET命令返回一个包含所有书籍的列表，而POST命令要求把额外的键book和一个JSON字符串作为Book对象表示的值。根据收到的消息，调用GetBooks或AddBook辅助方法。通过调用SendResponseAsync把从这些消息返回的结果返回给调用者：

```
private async void OnRequestReceived(AppServiceConnection sender,
  AppServiceRequestReceivedEventArgs args)
{
  AppServiceDeferral deferral = args.GetDeferral();
  try
  {
    ValueSet message = args.Request.Message;
    ValueSet result = null;
    switch (message["command"].ToString())
    {
      case "GET":
        result = GetBooks();
        break;
```

```
        case "POST":
          result = AddBook(message["book"].ToString());
        break;
          default:
        break;
      }
      await args.Request.SendResponseAsync(result);
    }
    finally
    {
      deferral.Complete();
    }
  }
```

GetBooks 方法使用 BooksRepository 获得 JSON 格式的所有书籍，它创建了一个 ValueSet，其键为 result：

```
private ValueSet GetBooks()
{
  var result = new ValueSet();
  result.Add("result", BooksRepository.Instance.Books.ToJson());
  return result;
}
```

AddBook 方法使用存储库添加一本书，并返回一个 ValueSet，其中的键是 result，值是 ok：

```
private ValueSet AddBook(string book)
{
  BooksRepository.Instance.AddBook(book.ToBook());
  var result = new ValueSet();
  result.Add("result", "ok");
  return result;
}
```

### 36.5.3　注册应用程序服务

现在需要通过操作系统注册应用程序服务。为此，创建一个正常 UWP 应用程序，它引用了 BooksCacheService。在此应用程序中，必须在 package.appxmanifest 中定义一个声明(见图 36-8)。在应用程序声明列表中添加一个应用程序服务，并指定名字。需要设置到后台任务的入口点，包括名称空间和类名。

对于客户端应用程序，需要 package.appxmanifest 定义的应用程序名和包名。为了查看包名，可以查看 PackageManifestEditor 的 Package 选项卡，也可以以编程方式调用 Package.Current.Id.FamilyName。为了便于在启动应用程序时查看这个名字，把它写入属性 PackageFamilyName，该属性绑定到用户界面的一个控件上(代码文件 AppServices/BooksCacheProvider/MainPage.xaml.cs)：

```
public sealed partial class MainPage: Page
{
  public MainPage()
  {
    this.InitializeComponent();
    PackageFamilyName = Package.Current.Id.FamilyName;
  }

  public string PackageFamilyName
  {
    get => (string)GetValue(PackageFamilyNameProperty);
    set => SetValue(PackageFamilyNameProperty, value);
  }

  public static readonly DependencyProperty PackageFamilyNameProperty =
    DependencyProperty.Register("PackageFamilyName", typeof(string),
      typeof(MainPage), new PropertyMetadata(string.Empty));
}
```

当运行这个应用程序时，它会注册后台任务，并显示客户端应用程序需要的包名。

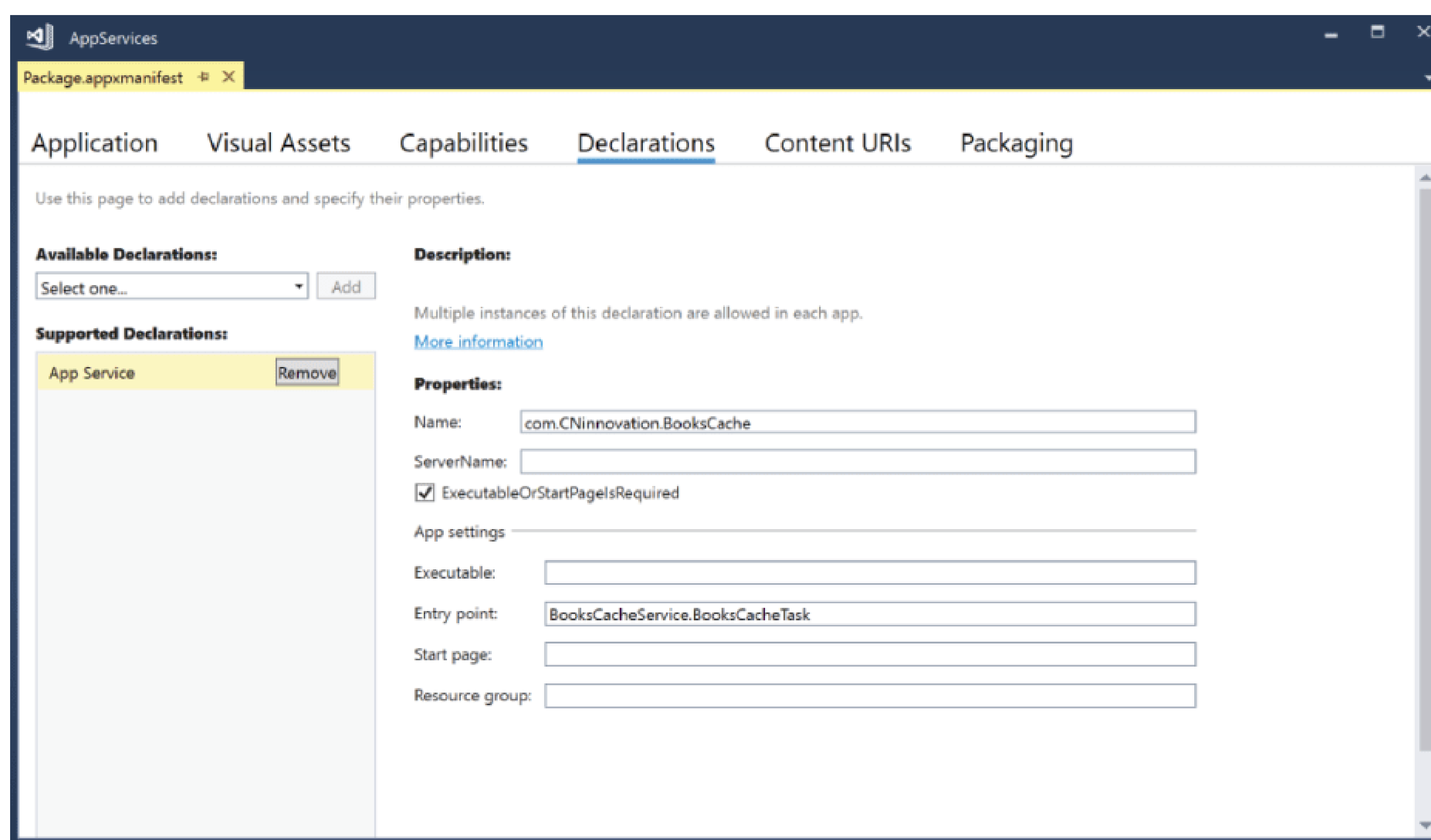



图 36-8

### 36.5.4 调用应用程序服务

在客户端应用程序中，现在可以调用应用程序服务。客户端应用程序 BooksCacheClient 的主要部分用视图模型实现。Books 属性绑定在 UI 中，显示从服务返回的所有书籍。这个集合用 GetBooksAsync 方法填充。GetBooksAsync 使用 GET 命令创建一个 ValueSet，使用 SendMessageAsync 辅助方法发送给应用程序服务。这个辅助方法返回一个 JSON 字符串，该字符串再转换为一个 Book 集合，用于填充 Books 属性的 ObservableCollection (代码文件 AppServices/BooksCacheClient/ViewModels/BooksViewModel.cs )：

```
public class BooksViewModel
{
  private const string BookServiceName = "com.CNinnovation.BooksCache";
  private const string BooksPackageName =
    "085f62ed-e72b-4c07-9970-b4d01c066dd6_p2wxv0ry6mv8g";

  public ObservableCollection<Book> Books { get; } =
    new ObservableCollection<Book>();

  public async void GetBooksAsync()
  {
    var message = new ValueSet();
    message.Add("command", "GET");
    string json = await SendMessageAsync(message);
    IEnumerable<Book> books = json.ToBooks();
    foreach (var book in books)
    {
      Books.Add(book);
    }
  }
  //...
}
```

PostBookAsync 方法创建了一个 Book 对象，序列化为 JSON，通过 ValueSet 把它发送给 SendMessageAsync 方法：

```
public string NewBookTitle { get; set; }
public string NewBookPublisher { get; set; }
public async void PostBookAsync()
{
  var message = new ValueSet();
  message.Add("command", "POST");
  string json = new Book
  {
```

```
    Title = NewBookTitle,
    Publisher = NewBookPublisher
  }.ToJson();
  message.Add("book", json);
  string result = await SendMessageAsync(message);
}
```

与应用程序服务相关的客户代码包含在 SendMessageAsync 方法中。其中创建了一个 AppServiceConnection。连接使用完后，通过 using 语句销毁，以关闭它。为了把连接映射到正确的服务上，需要提供 AppServiceName 和 PackageFamilyName 属性。设置这些属性后，通过调用方法 OpenAsync 来打开连接。只有成功地打开连接，才能在调用方法中发送请求和接收到的 ValueSet。AppServiceConnection 方法 SendMessageAsync 把请求发送给服务，返回一个 AppServiceResponse 对象。响应包含来自服务的结果，相应的处理如下：

```
private async Task<string> SendMessageAsync(ValueSet message)
{
  using (var connection = new AppServiceConnection())
  {
    connection.AppServiceName = BookServiceName;
    connection.PackageFamilyName = BooksPackageName;
    AppServiceConnectionStatus status = await connection.OpenAsync();
    if (status == AppServiceConnectionStatus.Success)
    {
      AppServiceResponse response =
        await connection.SendMessageAsync(message);
      if (response.Status == AppServiceResponseStatus.Success &&
        response.Message.ContainsKey("result"))
      {
        string result = response.Message["result"].ToString();
        return result;
      }
      else
      {
        await ShowServiceErrorAsync(response.Status);
      }
    }
    else
    {
      await ShowConnectionErrorAsync(status);
    }
    return string.Empty;
  }
}
```

在构建解决方案，部署提供程序和客户机应用程序后，就可以启动客户机应用程序来调用服务。还可以创建多个客户机应用程序，来调用相同的服务。

运行提供程序，注册后台任务后，可以运行客户端应用程序，获取图书，并添加新的图书，如图 36-9 所示。

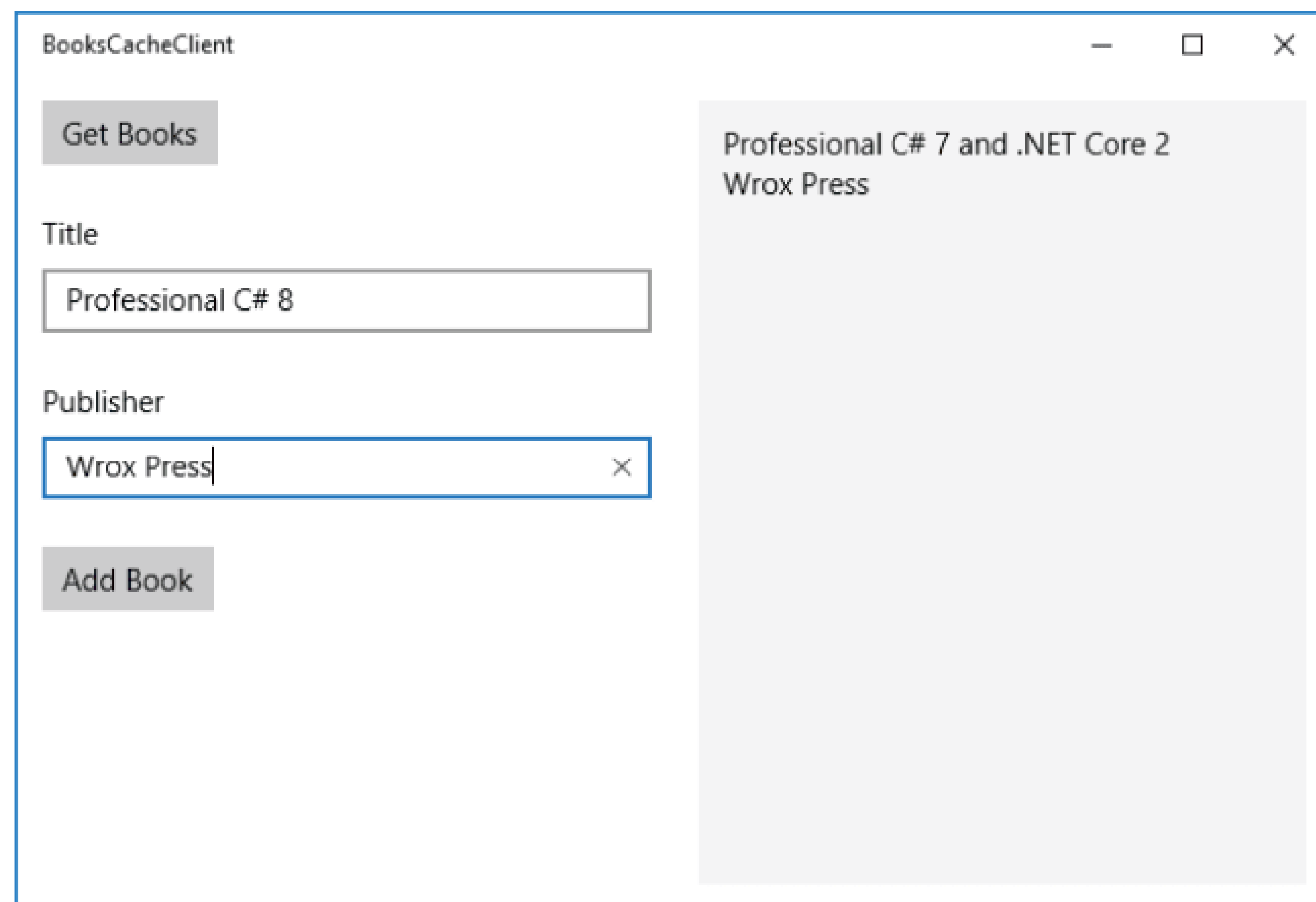



图 36-9

# 36.6　高级的编译绑定

第 33 章涵盖了与 Windows 应用程序的编译绑定，但编译绑定还有一些其他有趣的特性。本节介绍绑定生命周期、绑定到方法和阶段绑定。

## 36.6.1　已编译数据绑定的生命周期

通过已编译的数据绑定，C#代码从 XAML 文件的绑定中生成。还可以通过编程方式影响绑定的生命周期。下面从一个在用户界面中绑定的简单 Book 类型开始(代码文件 CompiledBindingLifetime/Models/Book.cs)：

```
public class Book : BindableBase
{
  public int BookId { get; set; }

  private string _title;
  public string Title
  {
    get => _title;
    set => Set(ref _title, value);
  }

  public string Publisher { get; set; }

  public override string ToString() => Title;
}
```

使用 page 类，创建一个只读属性 Book，返回一个 Book 实例。可以更改 Book 实例的值，而 Book 实例本身仅用于读取(代码文件 CompiledBindingLifetime/MainPage.xaml.cs)：

```
public Book Book { get; } = new Book
{
  Title = "Professional C# 7",
  Publisher = "Wrox Press"
};
```

在 XAML 代码中，Title 属性处于 TextBlock 的 Text 属性的 OneWay 模式，Publisher 被绑定而不指定模式，这意味着它被绑定 OneTime(代码文件 CompiledBindingLifetime/MainPage.xaml)：

```
<StackPanel>
  <TextBlock Text="{x:Bind Book.Title, Mode=OneWay}" />
  <TextBlock Text="{x:Bind Book.Publisher}" />
</StackPanel>
```

接下来，绑定几个 AppBarButton 控件，来更改已编译绑定的生命周期。一个按钮的 Click 事件绑定到 OnChangeBook 方法。这个方法使用调用的方式来更改图书的标题。如果尝试这样做，标题会因为进行 OneTime 绑定而立即更新(代码文件 CompiledBindingLifetime/MainPage.xaml.cs)：

```
private int csharpversion = 7;
public void OnChangeBook() =>
  Book.Title = $"Professional C# {++csharpversion}";
```

但是，可以停止对绑定的跟踪。使用页面的 Bindings 属性调用 StopTracking 方法(如果使用已编译绑定，则创建此属性)，将删除所有绑定侦听器。在调用 OnChangeBook 方法之前调用此方法时，该书的更新没有反映在用户界面中(代码文件 CompiledBindingLifetime/MainPage.xaml.cs)：

```
private void OnStopTracking() =>
  Bindings.StopTracking();
```

要从绑定源上显式更新用户界面，可以调用 update 方法。调用此方法不仅反映了 OneWay 绑定或 TwoWay 绑定的更改，还反映了 OneTime 绑定(代码文件 CompiledBindingLifetime/MainPage.xaml.cs)：

```
private void OnUpdateBinding() =>
  Bindings.Update();
```

在加载窗口时，将调用 Initialize 方法。这个方法调用一次 Update 方法。当再次调用 Initialize 时，不会再次调用 Update。需要直接调用方法 Update 进行显式更新。Update 和 StopTracking 是在初始化后用已编译绑定控

制生命周期的两个重要方法。还可以使用 Update 方法来更新那些没有实现 INotifyPropertyChanged 的属性。

运行应用程序时(见图 36-10)，可以单击下方区域的 Stop 按钮，然后单击 Edit 按钮更改图书。只有在显式单击 Refresh 按钮时才会发生更新。如果没有单击 Stop 按钮，则在单击 Edit 按钮时立即更新 UI。

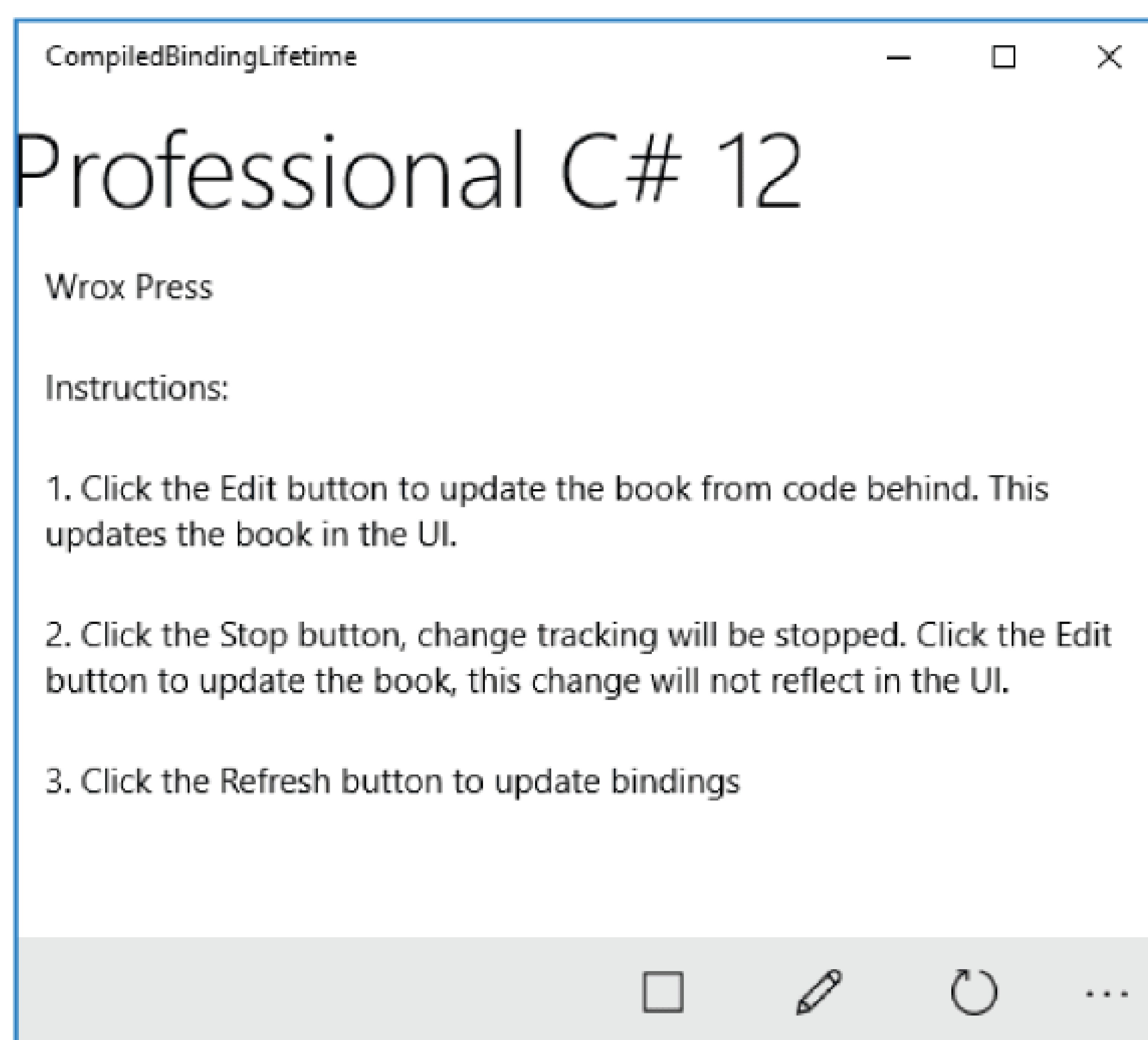



图 36-10

### 36.6.2　绑定到方法上

使用已编译绑定，就已经使用了事件绑定——把事件绑定到方法上。已编译绑定也支持把属性绑定到方法上。例如，可以使用它替换 WPF 支持的多绑定特性，将模型的多个属性绑定到 XAML 元素的一个属性。通过绑定到方法，还可以替换值转换器。

下一个示例应用程序使用绑定到方法。示例应用程序中使用的模型是 Person 类。这个类没有实现通知更改，所以需要在绑定中调用 Update 方法，来查看 UI 中更新的更改(代码文件 CompiledBindingMethods/Models/Person.cs)：

```
public class Person
{
  public string GivenName { get; set; }
  public string Surname { get; set; }
}
```

在 MainPage 中实例化两个 Person 对象，并分配给属性 Person1 和 Person2(代码文件 CompiledBindingMethods/MainPage.xaml.cs)：

```
public MainPage()
{
  Person1 = new Person { GivenName = "Katharina", Surname = "Nagel" };
  Person2 = new Person { GivenName = "Stephanie", Surname = "Nagel" };
  this.InitializeComponent();
}

public Person Person1 { get; }
public Person Person2 { get; set; }
```

在代码隐藏文件中，方法 ToName 将 Person 对象转换为字符串。实现两个将 Person 对象转换为字符串的不同变体：第一个变体返回姓之前给定的名称；第二个变体返回给定名称前的姓，中间有逗号。为了在变体之间进行选择，ToName 方法的第二个参数接受一个布尔输入值，来选择其中一个选项(代码文件 CompiledBindingMethods/MainPage.xaml.cs)：

```
public string ToName(Person p, bool firstLast)
{
  if (p == null) throw new ArgumentNullException(nameof(p));
```

```
    if (firstLast)
    {
      return $"{p.FirstName} {p.LastName}";
    }
    else
    {
      return $"{p.LastName}, {p.FirstName}";
    }
  }
```

要调用 ToName 方法，方法名与 x:Bind 标记扩展名一起使用。把两个值传递给 ToName 方法：在代码隐藏文件中定义的 Person1 属性值和一个布尔参数。x:True 和 x:False 在 http://schemas.microsoft.com/winfx/2006/xaml 名称空间中定义为 XAML (代码文件 CompiledBindingMethods/MainPage.xaml)：

```
<TextBlock Text="{x:Bind ToName(Person1, x:False)}" />
<TextBlock Text="{x:Bind ToName(Person1, x:True)}" />
```

使用这些绑定，Person1 对象显示为以下两种变体：

- FirstName LastName
- LastName，FirstName

把值改回来怎么样？还可以定义使用一个方法，该方法用 XAML 代码中的双向绑定来调用。属性的值传递给方法，需要定义一个方法来接收这个参数。绑定 TextBlock 的 Text 属性时，会收到一个字符串。ToPerson2 方法声明类型为 string 的参数。将此值拆分，将其值赋给 Person2 的 GivenName 和 Surname 属性。因为 GivenName 和 Surname 属性没有实现更改通知，所以显式调用 Update 方法，来更新用户界面(代码文件 CompiledBindingMethods/MainPage.xaml.cs)：

```
public void ToPerson2(string name)
{
  string[] names = name.Split(' ');
  if (names.Length != 2) return, // don't do anything with wrong inputs
  Person2.GivenName = names[0];
  Person2.Surname = names[1];
  Bindings.Update();
}
```

调用方法来接收 Text 属性值的方法使用了 x:Bind 标记扩展的 BindBack 属性。BindBack 需要方法的名称。另外，已编译的绑定需要声明为 TwoWay(代码文件 CompiledBindingMethods/MainPage. xaml)：

```
<TextBox
  Text="{x:Bind ToName(Person2, x:True), BindBack=ToPerson2, Mode=TwoWay}" />
<TextBlock Text="{x:Bind ToName(Person2, x:True)}" />
```

运行应用程序时，可以看到绑定值，并使用绑定到方法来更改 TextBox 中的值。

### 36.6.3 用 x:Bind 分阶段

用户不想等待。有时候获取信息需要一些时间。使用阶段化可以改善用户体验，因为可以更早地提供一些数据，并随着更多数据可用而更新用户界面。

> **注意：**
> *如果熟悉来自 WPF 的优先级绑定，那么使用阶段编译绑定可以提供与不同实现类似的功能。*

通过分阶段，可以通过使用 ListView 和 GridView 控件来定义不同的阶段。只有这些控件支持分阶段。

示例应用程序实现了一个典型的场景：从 API 服务中检索数据，这可能需要一些时间。同时，还应显示其他信息。数据是逐阶段检索的。对于每个阶段，返回的数据类型可能不同。示例应用程序最初只显示静态信息，但是类似的本地缓存信息可以用于显示。

从服务中获取的信息由 LunchMenu 类定义(代码文件 PhasedBinding/Models/LunchMenu.cs)：

```
public class LunchMenu
{
  public int MenuId { get; set; }
  public string Text { get; set; }
```

```
    public string ImageUrl { get; set; }
}
```

LunchMenuService 类模拟从一个服务中获取菜单列表。这里只是一个内存列表。从网络返回一个简单的列表取决于网络速度和到服务器的距离，且可能看不到这两个阶段之间的巨大差异。但是，需要的服务器端处理越多，效果就越容易看到。为了模拟更长的过程，从 GetLunchMenusAsync 方法中返回的菜单数据之前，需要延迟 10 秒 (代码文件 PhasedBinding/Services/LunchMenuService.cs.cs)：

```
public class LunchMenuService
{
  private IEnumerable<LunchMenu> _menusList = new List<LunchMenu>()
  {
    new LunchMenu { MenuId = 1, Text = "Chicken Salad", ImageUrl =
      "https://kantinem101.blob.core.windows.net/menuimages/Backhendelsalat"},
    new LunchMenu { MenuId = 2, Text = "Cordon Bleu", ImageUrl =
      "https://kantinem101.blob.core.windows.net/menuimages/CordonBleu_250"},
    new LunchMenu { MenuId = 3, Text = "Wiener Schnitzel", ImageUrl =
      "https://kantinem101.blob.core.windows.net/menuimages/" +
      "WienerSchnitzel_250"},
    new LunchMenu { MenuId = 4, Text = "Lasagne", ImageUrl =
      "https://kantinem101.blob.core.windows.net/menuimages/Lasagne_250"},
    new LunchMenu { MenuId = 5, Text = "Lentils with bacon and dumplings",
      ImageUrl =
        "https://kantinem101.blob.core.windows.net/menuimages/Linsen_250"},
    new LunchMenu { MenuId = 6, Text = "Schweinslungenbraten", ImageUrl =
      "https://kantinem101.blob.core.windows.net/menuimages/" +
      "Schweinslungenbraten_250"},
    new LunchMenu { MenuId = 7, Text = "Spätzle", ImageUrl =
      "https://kantinem101.blob.core.windows.net/menuimages/" +
      "Spinatspaetzle_250"},
    new LunchMenu { MenuId = 8, Text = "Topfennockerl", ImageUrl =
      "https://kantinem101.blob.core.windows.net/menuimages/" +
      "Topfennockerl_250"},
    new LunchMenu { MenuId = 9, Text = "Fried trout with potatoes", ImageUrl =
      "https://kantinem101.blob.core.windows.net/menuimages/forelle_250"},
  };

  public async Task<IEnumerable<LunchMenu>> GetLunchMenusAsync()
  {
    await Task.Delay(10000); // simulate a delay
    return _menusList;
  }
}
```

用户界面中的列表绑定到 LunchMenuViewModel 上。这个视图模型类定义了 LunchMenu 属性，它返回与其名称相同类型的对象。此外，还提供了 IntroText 属性。这个只读属性只返回静态文本。在 LunchMenu 的结果出来之前，将首先显示介绍文本(代码文件 PhasedBinding/ViewModels/LunchMenuViewModel.cs)：

```
public class LunchMenuViewModel : BindableBase
{
  private LunchMenuService _service = new LunchMenuService();

  public LunchMenuViewModel()
  {
  }

  public string IntroText { get; } = "A Lunch";

  private LunchMenu _lunchMenu;
  public LunchMenu LunchMenu
  {
    get => _lunchMenu;
    set => Set(ref _lunchMenu, value);
  }
}
```

在代码隐藏文件中，创建了一个 LunchViewModel 对象列表，并分配给 LunchViewModels 属性。在页面的

开头，这个视图模型的属性 IntroText 已经初始化并可以显示出来。加载页面之后，将调用 OnLoaded 方法。在此方法中，从服务中检索午餐菜单，并使用返回的结果更新 ObservableCollection 中现有的 LunchMenuViewModel 对象 (代码文件 PhasedBinding/MainPage.xaml cs)：

```
public sealed partial class MainPage : Page
{
  private ObservableCollection<LunchMenuViewModel> _lunchViewModels =
    new ObservableCollection<LunchMenuViewModel>(
      Enumerable.Range(0, 8).Select(x => new LunchMenuViewModel()));

  private LunchMenuService _service = new LunchMenuService();
  public ObservableCollection<LunchMenuViewModel> LunchViewModels =>
    _lunchViewModels;

  public MainPage()
  {
    this.InitializeComponent();
    this.Loaded += OnLoaded;
  }

  private async void OnLoaded(object sender, RoutedEventArgs e)
  {
    try
    {
      List<LunchMenu> lunchMenus =
        (await _service.GetLunchMenusAsync()).ToList();
      for (int i = 0; i < 8; i++)
      {
        lunchViewModels[i].LunchMenu = lunchMenus[i];
      }
    }
    catch (Exception ex)
    {
      // TODO: log the exception
      throw;
    }
  }
}
```

接下来重要的代码是 XAML 语法，ListView 控件绑定到在代码隐藏文件中定义的 LunchViewModels。要定义视图模型应该显示什么，使用一个 DataTemplate。在这个数据模板中，除了 x:Bind 标记表达式之外，还可以看到 x:Phase 属性。用 x:Phase 属性定义的数字指定了阶段排序。绑定是按照绑定阶段的顺序进行的。在示例代码的第一个阶段中，只显示了 IntroText 属性的值。还可以添加本地映像。第 2 阶段从视图模型的 LunchMenu 属性中显示文本信息，第 3 阶段显示图像。在 XAML 代码中验证阶段 2 和阶段 3 之间的不同订单。要在图像上显示文本，首先声明图像。第一行中的文本框只是为了演示在加载数据时，UI 是有响应的。可以输入一些文本，并在处理不同阶段时使用 UI (代码文件 PhasedBinding/MainPage.xaml)：

```
<TextBox Header="Enter Some Text" PlaceholderText=
  "UI is responsive - enter some text while the actual data is retrieved"
  Grid.Row="1" />
<ListView HorizontalAlignment="Center" x:Name="MenuItems"
  ItemsSource="{x:Bind LunchViewModels, Mode=OneWay}"
  SelectionMode="None" Grid.Row="2">
  <ListView.ItemTemplate>
    <DataTemplate x:DataType="vm:LunchMenuViewModel">
      <Grid Margin="12">
        <TextBlock Text="{x:Bind IntroText}" x:Phase="1" />
        <Image Width="300" Source="{x:Bind LunchMenu.ImageUrl, Mode=OneWay}"
          x:Phase="3" />
        <TextBlock Margin="8" VerticalAlignment="Bottom"
          HorizontalTextAlignment="Center"
          Text="{x:Bind LunchMenu.Text, Mode=OneWay}"
          Style="{StaticResource SubtitleTextBlockStyle}" FontWeight="Bold"
          x:Phase="2" />
      </Grid>
    </DataTemplate>
  </ListView.ItemTemplate>
</ListView>
```

运行该应用程序时，首先会看到图 36-11 中的初始数据，然后会看到使用菜单信息更新的数据和图 36-12 中的图像。

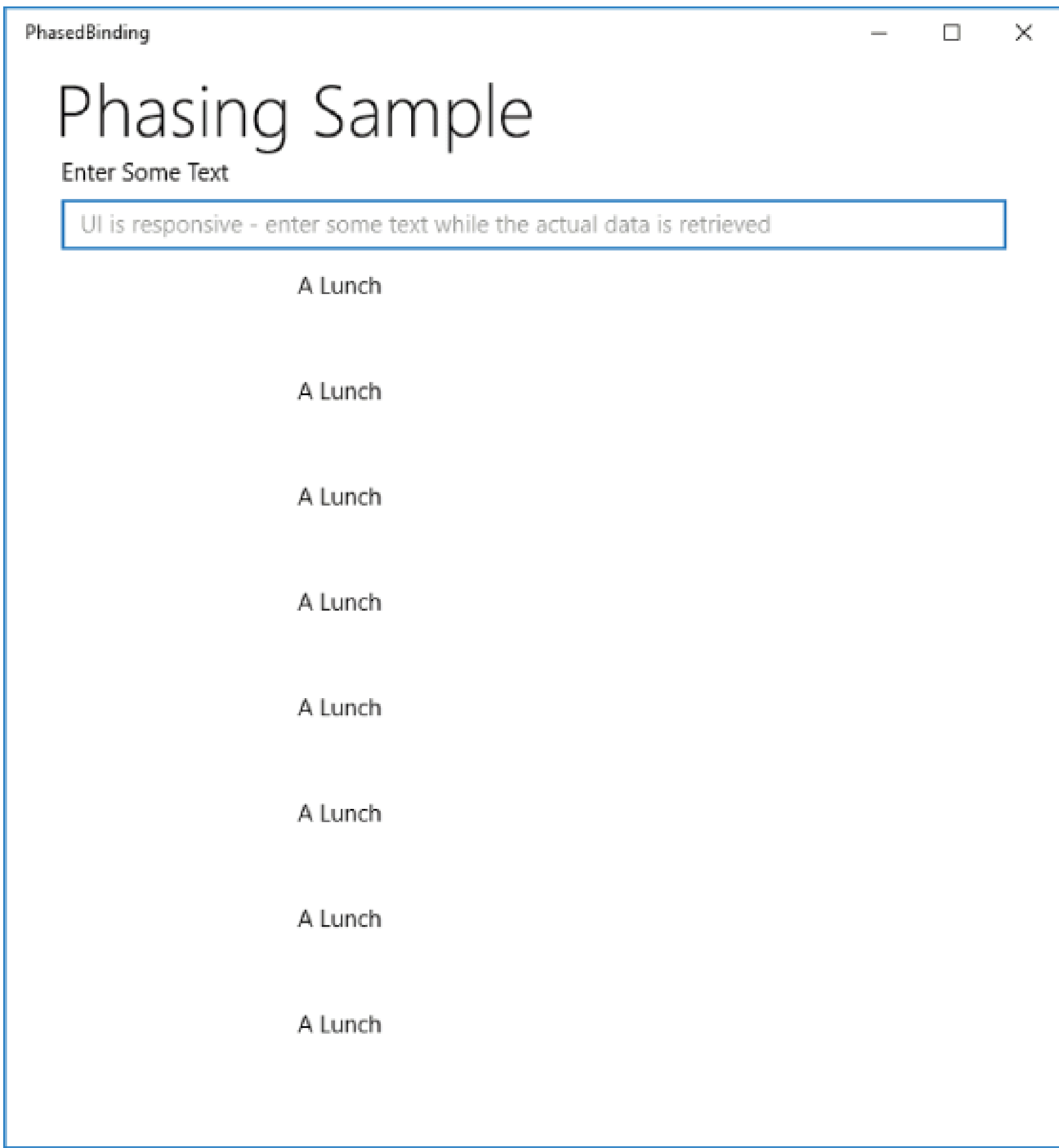



图 36-11

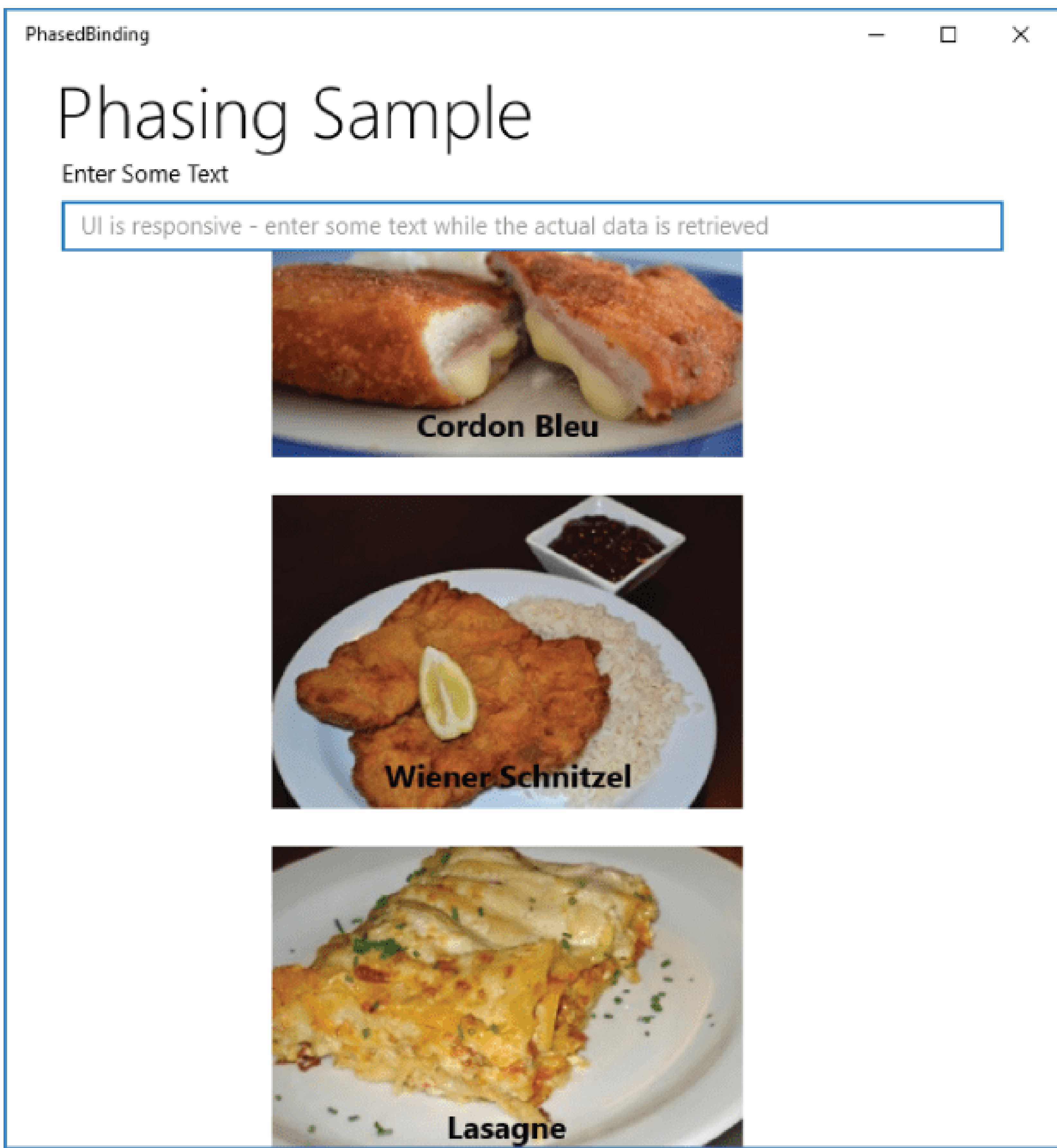



图 36-12

**注意：**
第 33 章说明了如何使用 CalendarView 控件实现各个阶段。

# 36.7 使用文本

Windows 应用程序对文本有丰富的支持。TextBlock 控件不仅支持简单字符串的显示，还支持更复杂的文本元素，比如使用不同的样式、权重、内联元素和块元素。RichTextBlock 控件扩展了这个功能，允许文本溢出。如果一列不够，可以很容易地将信息流到溢出区域。使用 RichTextBox 控件，支持 RTF(富文本文件)的使用。

## 36.7.1 使用字体

文本的一个重要方面是它的外观和字体的重要性。通过 TextBlock 控件，可以使用属性 FontWeight、FontStyle、FontStretch、FontSize 和 FontFamily 指定字体：

- FontWeight—— FontWeights 类指定的预定义值，它提供了如 ExtraLight、Light、Medium、Normal、Bold 和 ExtraBold 等值。
- FontStyle——FontStyles 类定义的值，它提供了 Normal、Italic 和 Oblique。
- FontStretch——使用它指定伸展字体的度(与正常长宽比相比)。FontStretch 定义了预定义的伸展度，范围是从 50%(UltraCondensed) 到 200%(UltraExpanded)。在该范围内的预定义值有 ExtraCondensed(62.5%)、Condensed(75%)、SemiCondensed(87.5%)、Normal(100%)、SemiExpanded(112.5%)、Expanded(125%)和 ExtraExpanded(150%)。
- FontSize——这是 double 类型，允许用与设备无关的单位指定字体的大小。
- FontFamily——用于指定首选的字体系列名，例如 Arial 或 Times New Roman。使用此属性，可以指定一个字体系列名称的列表，因此，如果一个字体不可用，则使用列表中的下一个字体。

为了了解不同字体的外观，下面的示例应用程序包括一个 ListView。ListView 显示在该字体列表中的字体名称。选择字体时，会显示更多的字体信息，如粗体字体的权重、字体样式的斜体、字体的展开和压缩，以及一些使用字体的文本(代码文件 FontsSample/MainPage.xaml)：

```
<ListView x:Name="listFonts" ItemsSource="{x:Bind FontNames, Mode=OneTime}"
  SelectedItem="{x:Bind SelectedFont, Mode=TwoWay}" Margin="12">
  <ListView.ItemTemplate>
    <DataTemplate x:DataType="x:String">
      <StackPanel Orientation="Horizontal">
        <TextBlock Text="{x:Bind Mode=OneTime}" FontFamily="{x:Bind }" />
      </StackPanel>
    </DataTemplate>
  </ListView.ItemTemplate>
</ListView>
<StackPanel Grid.Column="1" Margin="12" Padding="8">
  <TextBlock Text="{x:Bind SelectedFont, Mode=OneWay}"
    FontFamily="{x:Bind SelectedFont, Mode=OneWay}" />
  <TextBlock Text="Bold" FontFamily="{x:Bind SelectedFont, Mode=OneWay}"
    FontWeight="Bold" />
  <TextBlock Text="Italic" FontFamily="{x:Bind SelectedFont, Mode=OneWay}"
    FontStyle="Italic" />
  <TextBlock Text="Expanded" FontFamily="{x:Bind SelectedFont, Mode=OneWay}"
    FontStretch="Expanded" />
  <TextBlock Text="Condensed" FontFamily="{x:Bind SelectedFont, Mode=OneWay}"
    FontStretch="Condensed" />
  <TextBlock Text="The quick brown fox jumped over the lazy dogs"
    FontFamily="{x:Bind SelectedFont, Mode=OneWay}" />
  <TextBlock Text="&#xE700;&#xE701;&#xE702;"
    FontFamily="{x:Bind SelectedFont, Mode=OneWay}" />
  <TextBlock Text="&#x2467;&#x2468;&#x2469;"
    FontFamily="{x:Bind SelectedFont, Mode=OneWay}" />
</StackPanel>
```

在代码隐藏文件中，在 Windows 10 系统中保证可用的字体组合在一个集合中——其中的字体有的适合于标题和 UI 元素，如 Calibri、Consolas 和 Segoe UI；有的适合于大量的文本，如 Cambria 和 Courier New；有的适合于符号和图标，如 Segoe UI Emoji 和 Segoe MDL2 Assets；以及非拉丁字体。合并的字体分配给 FontNames 属性。此属性在 UI 中绑定。依赖属性用于选择字体(代码文件 FontsSample/MainPage.xaml.cs)：

```
public sealed partial class MainPage : Page
```

```
{
  public MainPage()
  {
    this.InitializeComponent();
    SelectedFont = FontNames.First();
  }

  // good for headings and UI elements
  private readonly string[] _sansSerifFontNames = { "Arial", "Calibri",
    "Consolas", "Segoe UI", "Segoe UI Historic", "Selawik", "Verdana" };

  // good for large amounts of text
  private readonly string[] _serifFontNames = { "Cambria", "Courier New",
    "Georgia", "Times New Roman" };

  // symbols and icons
  private readonly string[] _symbolsAndIconFontNames = { "Segoe MDL2 Assets",
    "Segoe UI Emoji", "Segoe UI Symbol" };

  // non-latin
  private readonly string[] _nonLatinFontNames = { "Ebrima", "Gadugi",
    "Javanese", "Leelawadee UI", "Malgun Gothic", "Microsoft Himalaya",
    "Microsoft JhengHei UI", "Microsoft PhagsPa", "Microsoft Tai Le",
    "Microsoft YaHei UI", "Microsoft Yi Baiti", "Mongolian Baiti", "MV Boli",
    "Myanmar Text", "Nirmala UI", "SimSun", "Yu Gothic", "Yu Gothic UI" };

  private string[] allFonts;
  public string[] FontNames => allFonts ?? (allFonts =
    _sansSerifFontNames.Concat(_serifFontNames)
      .Concat(_symbolsAndIconFontNames).Concat(_nonLatinFontNames).ToArray());

  public string SelectedFont
  {
    get => (string)GetValue(SelectedFontProperty);
    set => SetValue(SelectedFontProperty, value);
  }

  public static readonly DependencyProperty SelectedFontProperty =
    DependencyProperty.Register("SelectedFont", typeof(string),
      typeof(MainPage), new PropertyMetadata(string.Empty));
}
```

在运行应用程序时，会得到具有不同特征的字体列表，如图 36-13 所示。

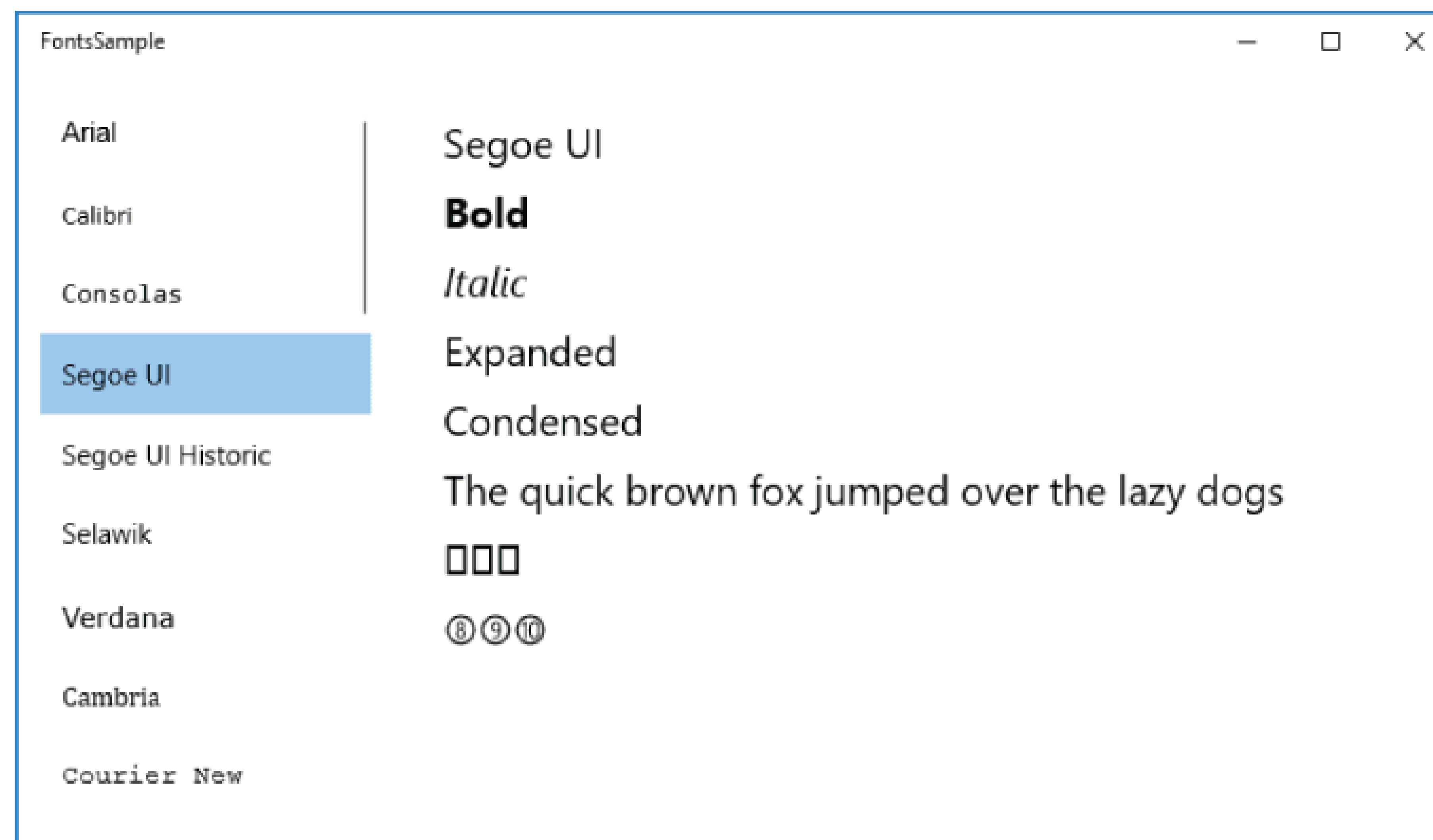



图 36-13

**注意：**

并非所有字体都定义了所有字符。有些文本可能不会使用 Segoe MDL2 *Assets* 显示，而这个字体在 Unicode 字符中显示图标，而其他字体没有定义元素。显示或未显示的特殊字符可以根据系统上安装的字体扩展名而有所不同。

### 36.7.2 内联和块元素

在定义文本元素时，需要区分从 Inline 类中派生的元素和从 Block 类中派生的元素。Inline 派生自基类 TextElement:，所有 Inline 元素都是 TextElement，并表示一个文本。Block 元素允许定义段落。Paragraph 类派生自 Block。块可以包含 Inline 元素。

看看下面包含 Bold、LineBreak、Hyperlink、Italic、Underline、Run 和 Span 元素的 TextBlock 元素(代码文件 TextSample/MainPage.xaml)：

```
<TextBlock Margin="12" x:Name="text1" IsTextSelectionEnabled="True"
  SelectionHighlightColor="Green" FontFamily="Segoe UI" Foreground="Black">
  <Bold>This is bold.</Bold><LineBreak />
  <Hyperlink NavigateUri="https://csharp.christiannagel.com">
    C# Blog</Hyperlink><LineBreak />
  <Italic>This is italic.</Italic><LineBreak />
  <Underline>This is underlined.</Underline><LineBreak />
  <Run>Run element</Run><LineBreak />
  <Span>Span element</Span><LineBreak />
  <Span FontFamily="Calibri">
    <Run FontSize="24">A span can contain inlines</Run>
    <Italic>Italic is a span<LineBreak />
      and thus <Underline>underlines</Underline> can contain inlines as well
  </Span>
</TextBlock>
```

所有定义为 TextBlock 子元素的元素都添加到 Inlines 属性中。类 TextBlock 将属性 Inlines 定义为 ContentProperty，就像 ButtonBase 类将属性 Content 定义为 ContentProperty 一样。

类 LineBreak、Run 和 Span 派生自 Inline，可用于 Inline 集合。Run 类定义了 Text 属性，可以用文本内容填充该属性，并使用 Run 元素的属性设置字体信息。Span 元素本身可以作为进一步内联元素的容器。Bold、Italic 和 Hyperlink 派生自 Span，因此，它们也可以用作 Inline 元素，它们还可以包含其他 Inline 元素。

图 36-14 显示了包含所有这些 Inline 元素的 TextBlock 元素。

图 36-14

RichTextBlock 给其内容使用 Blocks 属性。Block 类定义文本格式属性 LineHeight、LineStackingStrategy、Margin 和 TextAlignment。Paragraph 是派生自 Block 的唯一具体类，可以与 Blocks 属性一起使用。Block 类不是抽象的，但是构造函数是用 protected 访问修饰符声明的。

下面的代码片段在 RichTextBlock 中使用了多个 Paragraph 元素。Paragraph 本身包含一个 Inline 元素列表(代码文件 TextSample/MainPage.xaml)：

```
<RichTextBlock>
  <Paragraph FontSize="18">Paragraph 1</Paragraph>
  <Paragraph LineStackingStrategy="BaselineToBaseline" LineHeight="16"
    TextAlignment="Justify">
    <Run FontStretch="ExtraExpanded" FontWeight="Black" Text="Paragraph 2" />
    <LineBreak />
    <Run>
      The quick brown fox jumped over the lazy dogs down 1234567890 times.
    </Run>
    <Run>
      The quick brown fox jumped over the lazy dogs down 1234567890 times.
    </Run>
  </Paragraph>
  <Paragraph>
    <Bold>Paragraph 3</Bold>
    <LineBreak />
    <InlineUIContainer>
      <Ellipse Width="30" Height="20" Fill="Red" />
    </InlineUIContainer>
    <LineBreak />
    <Run>More Text</Run>
  </Paragraph>
  <Paragraph LineStackingStrategy="BaselineToBaseline" LineHeight="16"
    TextAlignment="Left">
```

```
    <Run FontWeight="Bold" Text="Paragraph 4" />
    <LineBreak />
    <Run>
      The quick brown fox jumped over the lazy dogs down 1234567890 times.
    </Run>
    <Run>
      The quick brown fox jumped over the lazy dogs down 1234567890 times.
    </Run>
  </Paragraph>
</RichTextBlock>
```

运行该应用程序时，可以看到图 36-15 中的 RichTextBlock。如果比较第 2 和第 4 段，就可以看到文本对齐方式的区别。第 2 段是居中对齐，第 4 段是左对齐。第 3 段包含一个 InlineUIContainer。InlineUIContainer 是另一个 Inline 元素，注意，这个 Inline 元素不能与 TextBlock 一起使用。它在 RichTextBlock 中成功显示。InlineUIContainer 元素可以显示其他 XAML 元素。示例代码在 RichTextBlock 中显示 Ellipse——这是一个 Shape 类。

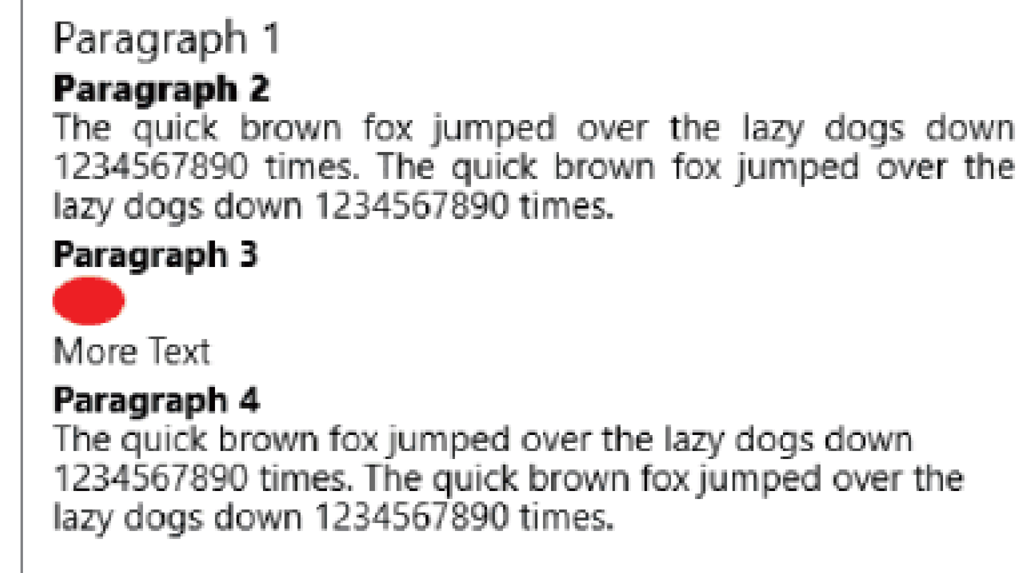



图 36-15

### 36.7.3　使用溢出区域

与 TextBlock 相比，RichTextBlock 可以显示多个段落。另一种可能是如果文本放不下，就允许 RichTextBlock 将文本溢出到另一个区域。这一次，文本是通过编程创建的。GetInlineElements 方法返回一个内联元素列表(代码文件 TextOverflow/MainPage.xaml.cs)：

```
public IEnumerable<Inline> GetInlineElements()
{
  var inlines = new List<Inline>();
  var header = new Bold();
  header.Inlines.Add(new Run { Text = "Lorem ipsum" });
  inlines.Add(header);
  inlines.Add(new LineBreak());
  inlines.Add(new Run { Text = "Lorem ipsum dolor sit amet, con...  "});
  inlines.Add(new LineBreak());
  return inlines;
}
```

GetBlock 方法返回包含内联元素的段落(代码文件 TextOverflow/MainPage.xaml.cs)：

```
public Block GetBlock()
{
  var paragraph = new Paragraph { TextAlignment = TextAlignment.Justify };
  foreach (var inline in GetInlineElements())
  {
    paragraph.Inlines.Add(inline);
  }
  return paragraph;
}
```

当加载页面时，从 GetBlock 方法获取的 Block 被反复添加到 RichTextBlock(代码文件 TextOverflow/MainPage.xaml.cs)：

```
public MainPage()
{
  InitializeComponent();
  Loaded += OnLoadData;
}

private void OnLoadData(object sender, RoutedEventArgs e)
{
  for (int i = 0; i < 8; i++)
  {
    textBlock.Blocks.Add(GetBlock());
  }
}
```

在用户界面中，两个 RichTextBlockOverflow 控件靠近 RichTextBlock。在 RichTextBlock 中，溢出的内容进

入第一个 RichTextBlockOverflow。这是由 OverflowContentTarget 属性定义的。该属性用于将内容从第一个 RichContentBlockOverflow 控件溢出到第二个控件(代码文件 TextOverflow/MainPage.xaml):

```
<RichTextBlock x:Name="textBlock" TextWrapping="Wrap" Margin="20,0"
  OverflowContentTarget="{x:Bind overflowContainer1}"
  TextLineBounds="Full"></RichTextBlock>
<RichTextBlockOverflow Visibility="Collapsed" x:Name="overflowContainer1"
  Grid.Column="1" Margin="20,0"
  OverflowContentTarget="{x:Bind overflowContainer2}"></RichTextBlockOverflow>
<RichTextBlockOverflow Visibility="Collapsed" x:Name="overflowContainer2"
  Grid.Column="2" Margin="20,0"></RichTextBlockOverflow>
```

如果页面太小，为了自动隐藏溢出控件，RichTextBlockOverflow 控件位于网格的列中(代码文件 TextOverflow/MainPage.xaml):

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition />
    <ColumnDefinition x:Name="Column2" />
    <ColumnDefinition x:Name="Column3" />
  </Grid.ColumnDefinitions>

  <!-- RichTextBlock and overflow controls -->
</Grid>
```

在 AdaptiveTrigger 的帮助下，溢出容器被设置为可见或压缩。AdaptiveTrigger 还定义了网格列的大小——其范围是 0 到星型大小。对于星型大小，RichTextBlock 和 RichTextBlockOverflow 具有相同的宽度(代码文件 TextOverflow/MainPage.xaml):

```
<VisualStateManager.VisualStateGroups>
  <VisualStateGroup>
    <VisualState x:Name="WideState">
      <VisualState.StateTriggers>
        <AdaptiveTrigger MinWindowWidth="1024" />
      </VisualState.StateTriggers>

      <VisualState.Setters>
        <Setter Target="overflowContainer1.Visibility" Value="Visible" />
        <Setter Target="overflowContainer2.Visibility" Value="Visible" />
        <Setter Target="Column3.Width" Value="*" />
        <Setter Target="Column2.Width" Value="*" />
      </VisualState.Setters>
    </VisualState>

    <VisualState x:Name="MediumState">
      <VisualState.StateTriggers>
        <AdaptiveTrigger MinWindowWidth="720" />
      </VisualState.StateTriggers>

      <VisualState.Setters>
        <Setter Target="overflowContainer1.Visibility" Value="Visible" />
        <Setter Target="overflowContainer2.Visibility" Value="Collapsed" />
        <Setter Target="Column3.Width" Value="0" />
        <Setter Target="Column2.Width" Value="*" />
      </VisualState.Setters>
    </VisualState>

    <VisualState x:Name="NarrowState">
      <VisualState.StateTriggers>
        <AdaptiveTrigger MinWindowWidth="320" />
      </VisualState.StateTriggers>

      <VisualState.Setters>
        <Setter Target="overflowContainer1.Visibility" Value="Collapsed" />
        <Setter Target="overflowContainer2.Visibility" Value="Collapsed" />
        <Setter Target="Column3.Width" Value="0" />
        <Setter Target="Column2.Width" Value="0" />
      </VisualState.Setters>
    </VisualState>
  </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

> **注意：**
> 自适应触发器参见第 33 章。

运行应用程序时，可以看到，宽屏幕有两个溢出控件的溢出(参见图 36-16)，中等屏幕有一个溢出控件(参见图 36-17)，窄屏幕没有溢出控件(参见图 36-18)。

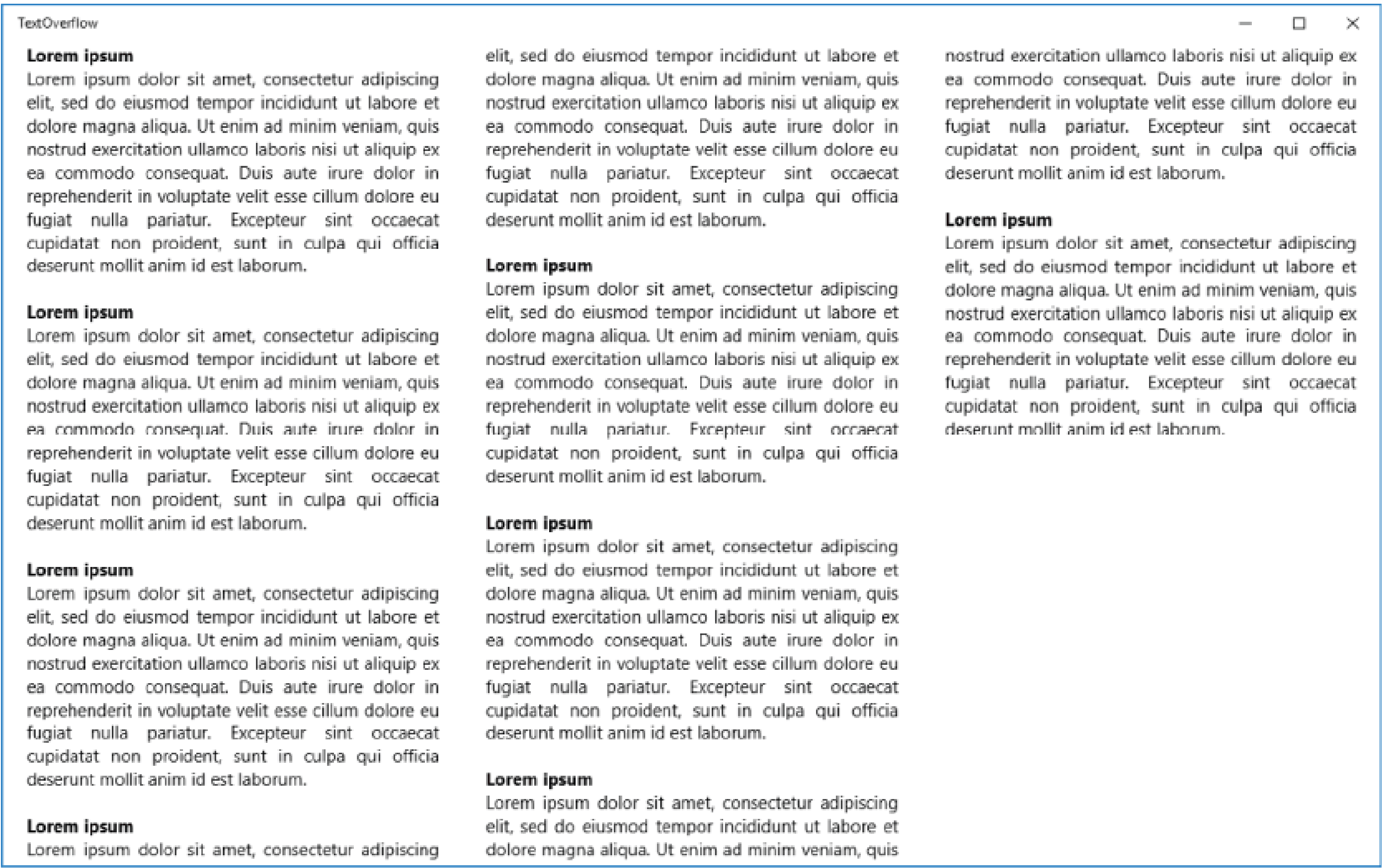



图 36-16




图 36-17

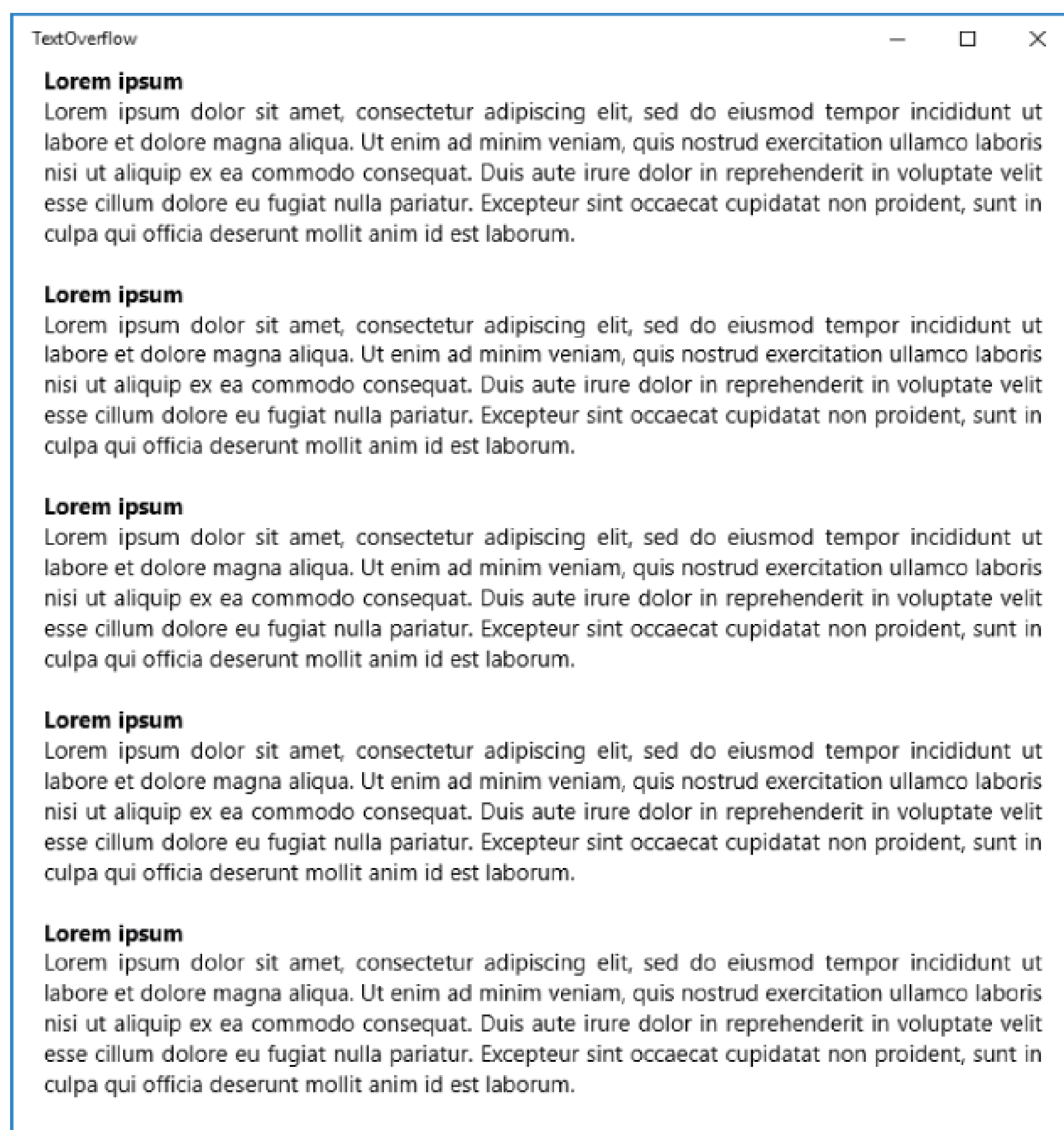



图 36-18

# 36.8　上墨

越来越多的 Windows 10 设备在使用钢笔。对于钢笔来说，上墨是一个重要的概念，Windows 应用程序很容易支持它。用户只需要使用 InkCanvas 控件来进行一些绘图。该控件支持使用钢笔、触摸屏和鼠标来上墨，还支持检索所有已创建的笔触，从而可以保存该信息(代码文件 InkSample/MainPage.xaml)：

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
  <InkCanvas x:Name="inkCanvas" Margin="8,32,8,8" />
  <!-- ... -->
```

默认情况下，InkCanvas 控件配置为支持钢笔。还可以通过设置 InkPresenter 的 InputDevicesType 属性，来定义它来支持鼠标和触摸屏(代码文件 InkSample/MainPage.xaml.cs)：

```
public MainPage()
{
  this.InitializeComponent();
  inkCanvas.InkPresenter.InputDeviceTypes = CoreInputDeviceTypes.Mouse |
    CoreInputDeviceTypes.Touch | CoreInputDeviceTypes.Pen;
    ColorSelection = new ColorSelection(inkCanvas);
  }
```

有了 InkCanvas，就可以使用输入设备，使用黑笔创建相同的绘图。要更改钢笔配置，还有一个 InkToolBar。这个工具栏需要使用 TargetInkCanvas 属性与 InkCanvas 关联。在示例应用程序中，另一个位于 InkToolbar 右侧的工具栏使用墨水笔触打开并保存文件(代码文件 InkSample/MainPage.xaml)：

```
<RelativePanel VerticalAlignment="Top">
  <InkToolbar x:Name="inkToolbar" RelativePanel.AlignLeftWithPanel="True"
    RelativePanel.AlignTopWithPanel="True"
    TargetInkCanvas="{x:Bind inkCanvas}" />
  <CommandBar RelativePanel.RightOf="inkToolbar"
    Template="{StaticResource CommandBarControlTemplate1}">
    <AppBarButton Icon="OpenFile" IsCompact="True" Click="{x:Bind OnLoad}"
      ToolTipService.ToolTip="Open File" />
    <AppBarButton Icon="Save" IsCompact="True" Click="{x:Bind OnSave}"
```

```
        ToolTipService.ToolTip="Save" />
      <AppBarButton Icon="Clear" IsCompact="True" Click="{x:Bind OnClear}"
        ToolTipService.ToolTip="Clear" />
    </CommandBar>
  </RelativePanel>
```

InkToolBar 提供自定义钢笔的颜色和大小的按钮，如图 36-19 所示。还可以选择标尺(参见图 36-20)和量角器(参见图 36-21)。

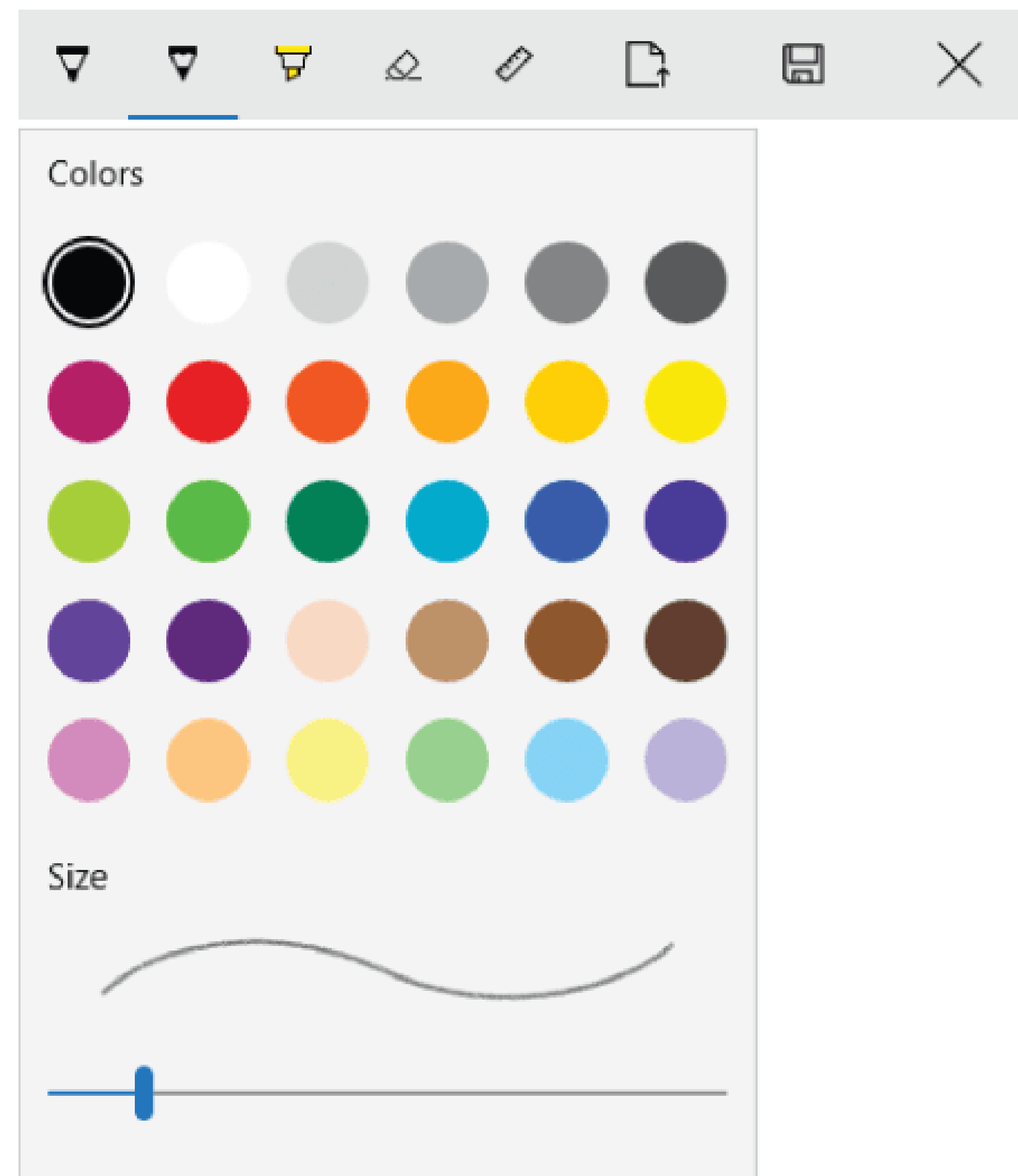



图 36-19

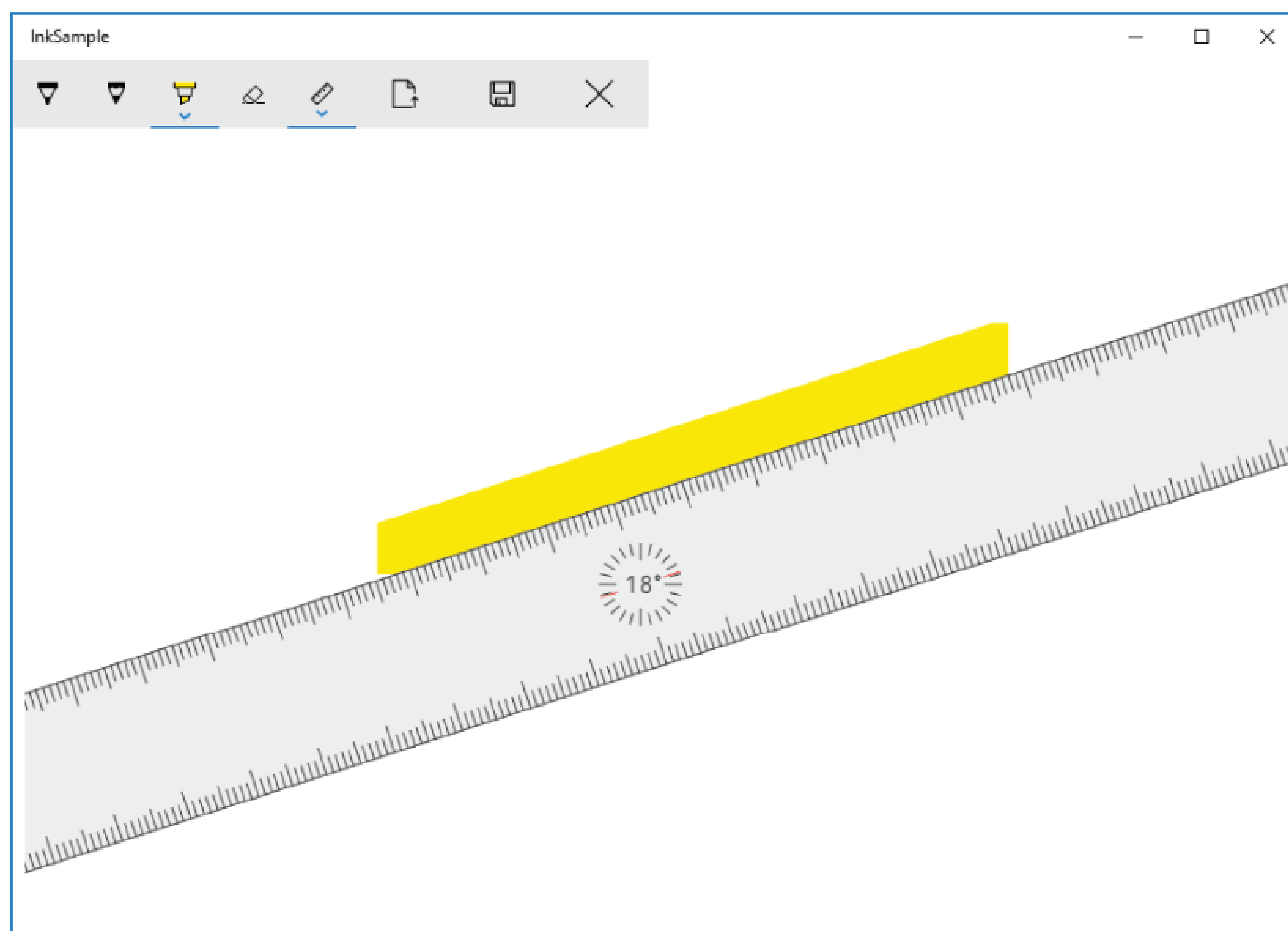



图 36-20

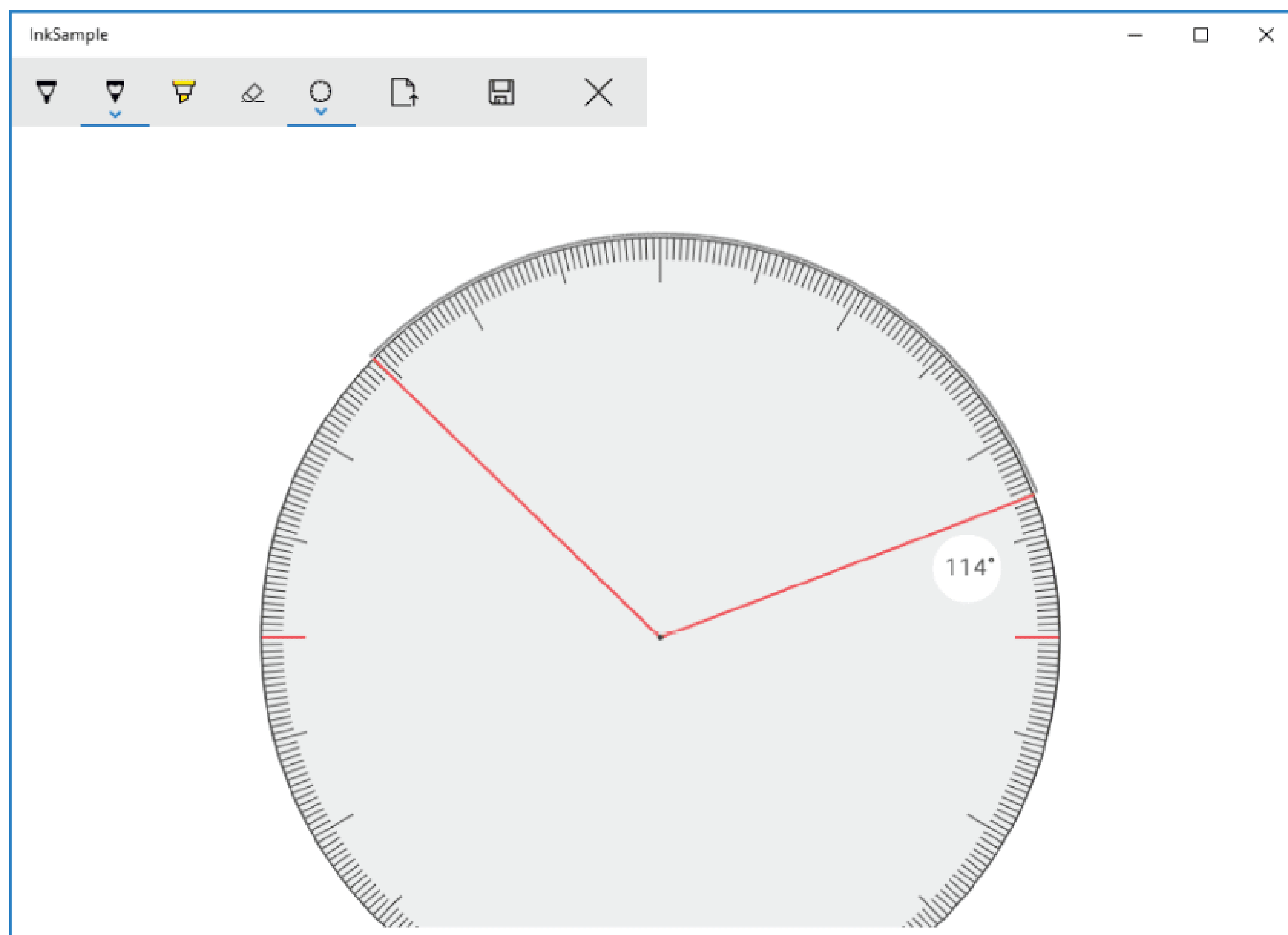



图 36-21

如前所述，InkCanvas 控件还支持访问已创建的笔触。下面的示例使用这些笔触将其存储在一个文件中。使用 FileSavePicker 选择文件。当用户单击先前创建的 Save AppBarButton 时，将调用 OnSave 方法。首先，通过分配启动位置、文件类型扩展名和文件名来配置 FileSavePicker。至少需要添加一个文件类型选项，以允许用户选择文件类型。调用 PickSaveFileAsync()方法时，将要求用户选择一个文件。该文件通过调用 OpenTransactedWriteAsync()方法来打开事务写入访问。InkCanvas 的笔触保存在 InkPresenter 的 StrokeContainer 下。笔触可以通过 SaveAsync 方法直接保存到流中(代码文件 InkSample/MainPage.xaml.cs)：

```
private const string FileTypeExtension = ".strokes";
public async void OnSave()
{
  var picker = new FileSavePicker
  {
    SuggestedStartLocation = PickerLocationId.PicturesLibrary,
    DefaultFileExtension = FileTypeExtension,
    SuggestedFileName = "sample"
  };

  picker.FileTypeChoices.Add("Stroke File", new List<string>()
  { FileTypeExtension });

  StorageFile file = await picker.PickSaveFileAsync();
  if (file != null)
  {
    using (StorageStreamTransaction tx = await file.OpenTransactedWriteAsync())
    {
      await inkCanvas.InkPresenter.StrokeContainer.SaveAsync(tx.Stream);
      await tx.CommitAsync();
    }
  }
}
```

**注意：**
使用 FileOpenPicker 和 FileSavePicker 来读写流详见第 22 章。

要加载文件，可以通过 LoadAsync 方法使用 FileOpenPicker 和 StrokeContainer(代码文件 InkSample/MainPage.xaml.cs)：

```
public async void OnLoad()
{
  var picker = new FileOpenPicker
  {
    SuggestedStartLocation = PickerLocationId.PicturesLibrary
  };

  picker.FileTypeFilter.Add(FileTypeExtension);
  StorageFile file = await picker.PickSingleFileAsync();
  if (file != null)
  {
    using (var stream = await file.OpenReadAsync())
    {
      await inkCanvas.InkPresenter.StrokeContainer.LoadAsync(stream);
    }
  }
}
```

运行应用程序时，如图 36-22 所示，很容易使用钢笔创建绘图。如果没有笔，也可以用手指触摸设备或使用鼠标，因为 InputDeviceTypes 属性已经做了相应的配置。

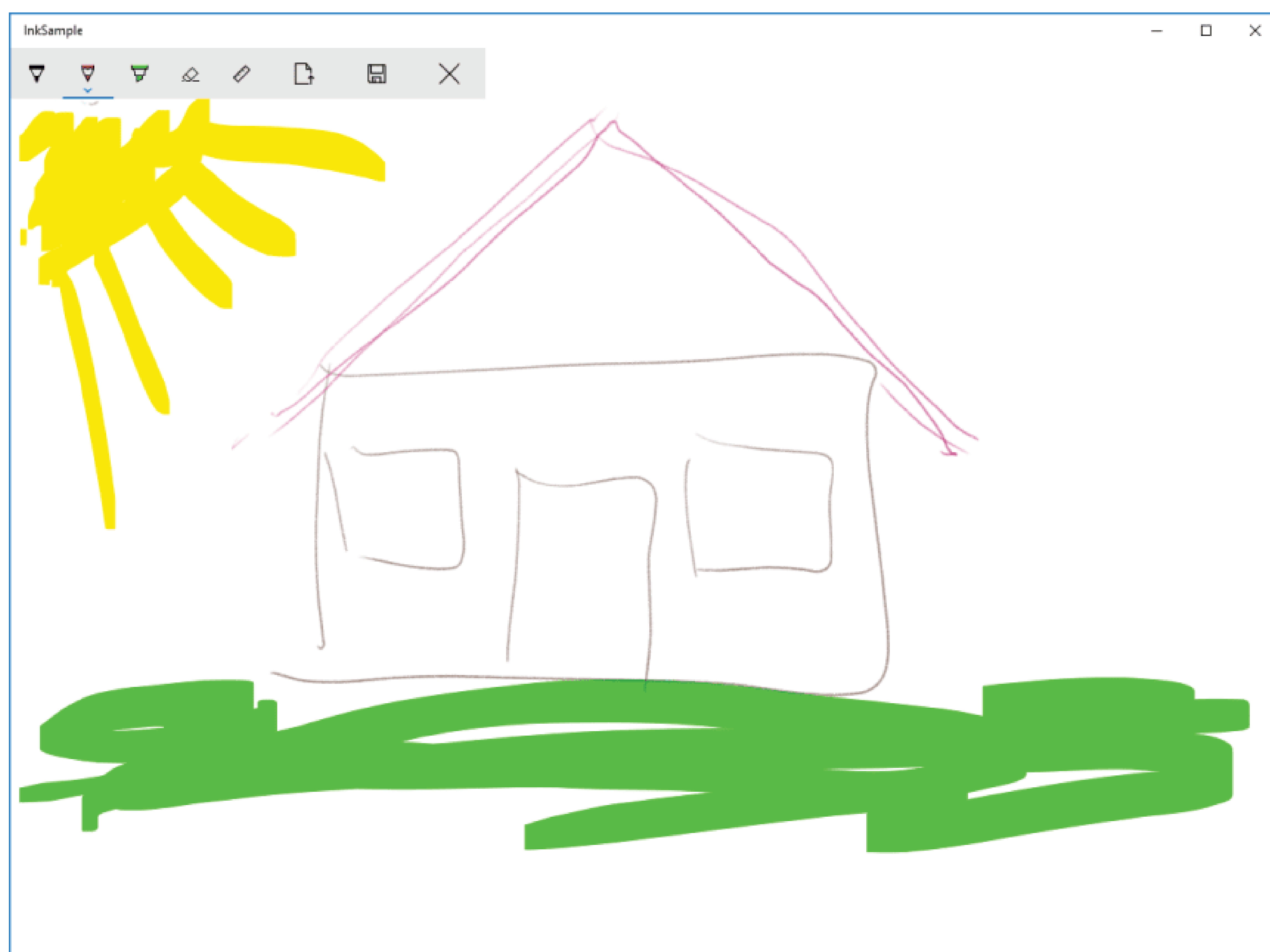



图 36-22

## 36.9 自动建议

Windows 的搜索功能有一段时间的历史了。在 Windows 8 中，搜索功能位于用户需要滑动才能打开的功能区上。在 Windows 10 中，它被 SearchBox 控件所取代。现在，SearchBox 被 AutoSuggestBox 控件所取代。该控件允许用户在控件中输入内容时，向用户提供建议。

这个控件有三个重要事件。一旦用户在控件中输入内容，TextChanged 事件就会被触发。在示例代码中，调用 OnTextChanged 处理程序方法。如果向用户提供了建议，而用户选择了该建议，就会触发 SuggestionChosen 事件。在用户输入文本(可能是建议或键入的其他单词)之后，就触发 QuerySubmitted 事件(代码文件 AutoSuggestSample/MainPage.xaml)：

```
<AutoSuggestBox Header="Formula 1 Champion"
  TextChanged="{x:Bind OnTextChanged}"
```

```
  SuggestionChosen="{x:Bind OnSuggestionChosen}"
  QuerySubmitted="{x:Bind OnQuerySubmitted}" />
```

为了用一些示例代码创建建议，请使用 HttpClient 类从 http://www.cninnovation.com/downloads/Racers.xml 中加载包含方程式 1 冠军的 XML 文件。导航到页面，检索 XML 文件，将内容转换为 Racer 对象列表(代码文件 AutoSuggestSample/MainPage.xaml.cs)：

```
private const string RacersUri =
  "http://www.cninnovation.com/downloads/Racers.xml";
private IEnumerable<Racer> _racers;

protected async override void OnNavigatedTo(NavigationEventArgs e)
{
  base.OnNavigatedTo(e);
  XElement xmlRacers = null;
  using (var client = new HttpClient())
  using (Stream stream = await client.GetStreamAsync(RacersUri))
  {
    xmlRacers = XElement.Load(stream);
  }

  _racers = xmlRacers.Elements("Racer").Select(r => new Racer
  {
    FirstName = r.Element("Firstname").Value,
    LastName = r.Element("Lastname").Value,
    Country = r.Element("Country").Value
  }).ToList();
}
```

Racer 类包含 FirstName、LastName、Country 属性和 ToString 方法的重载(代码文件 AutoSuggestSample/Models/Racer.cs)：

```
public class Racer
{
  public string FirstName { get; set; }
  public string LastName { get; set; }
  public string Country { get; set; }
  public override string ToString() => $"{FirstName} {LastName}, {Country}";
}
```

只要 AutoSuggestBox 的文本发生变化，就会调用 OnTextChanged 事件。接收的参数是 AutoSuggestBox 本身(发送者)和 AutoSuggestBoxTextChangedEventArgs。使用 AutoSuggestBoxTextChangedEventArgs，发生变化的原因显示在 Reason 属性中。可能的原因包括 UserInput、ProgrammaticChange 和 SuggestionChosen。只有原因是 UserInput，才需要向用户提供建议。这里还进行了检查，以查看用户是否输入了至少两个字符。通过访问 AutoSuggestBox 的 Text 属性来检索用户输入。此文本基于输入字符串查询名字、姓氏和国家。查询的结果分配给 AutoSuggestBox 的 ItemsSource 属性(代码文件 AutoSuggestSample/MainPage.xaml.cs)：

```
private void OnTextChanged(AutoSuggestBox sender,
  AutoSuggestBoxTextChangedEventArgs args)
{
  if (args.Reason == AutoSuggestionBoxTextChangeReason.UserInput &&
    sender.Text.Length >= 2)
  {
    string input = sender.Text;
    var racers = _racers.Where(
      r => r.FirstName.StartsWith(input,
        StringComparison.CurrentCultureIgnoreCase))
        .OrderBy(r => r.FirstName).ThenBy(r => r.LastName)
        .ThenBy(r => r.Country).ToArray();

    if (racers.Length == 0)
    {
      racers = _racers.Where(r => r.LastName.StartsWith(input,
        StringComparison.CurrentCultureIgnoreCase))
        .OrderBy(r => r.LastName).ThenBy(r => r.FirstName)
        .ThenBy(r => r.Country).ToArray();

      if (racers.Length == 0)
      {
        racers = _racers.Where(r => r.Country.StartsWith(input,
```

```
            StringComparison.CurrentCultureIgnoreCase))
            .OrderBy(r => r.Country).ThenBy(r => r.LastName)
            .ThenBy(r => r.FirstName).ToArray();
        }
      }

    sender.ItemsSource = racers;
    }
  }
```

运行应用程序，在 AutoSuggestBox 中输入 Aus 时，该查询无法从该文本中找到名字或姓氏，但找到了国家。建议列表中显示了来自以 Aus 开头的国家的一级方程式冠军，如图 36-23 所示。

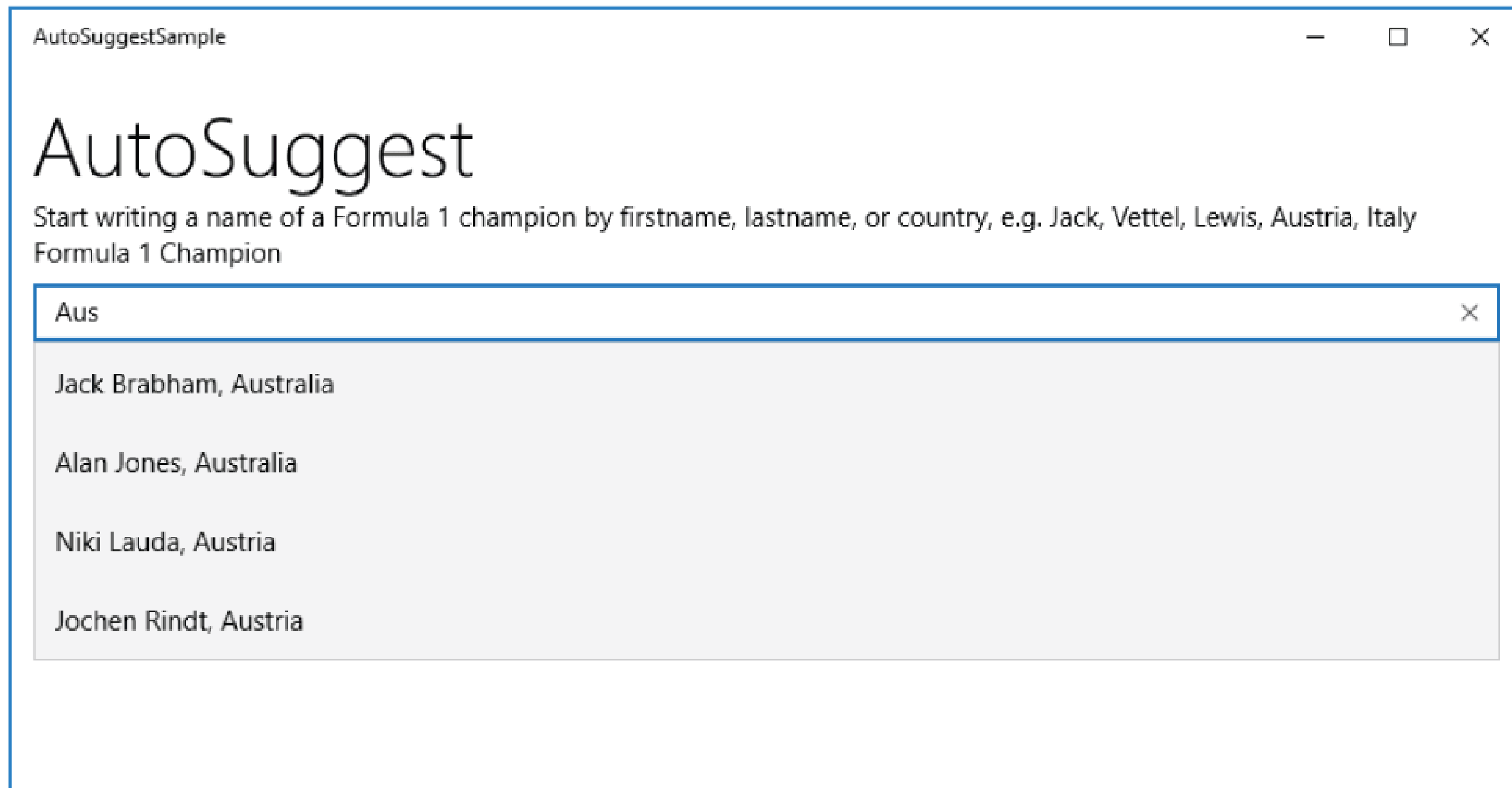



图 36-23

如果用户选择了其中一个建议，将调用 OnSuggestionChosen 处理程序。建议可以从 AutoSuggestBox-SuggestionChosenEventArgs 的 SelectedItem 属性中检索：

```
private async void OnSuggestionChosen(AutoSuggestBox sender,
  AutoSuggestBoxSuggestionChosenEventArgs args)
{
  var dlg = new MessageDialog($"suggestion: {args.SelectedItem}");
  await dlg.ShowAsync();
}
```

无论用户是否选择了建议，都会调用 onQuerySubmit 方法来显示结果。结果显示在 AutoSuggest Box Query SubmittedEventArgs 参数的 QueryText 属性中。如果选择了建议，结果可以在 ChosenSuggestion 属性中找到：

```
private async void OnQuerySubmitted(AutoSuggestBox sender,
AutoSuggestBoxQuerySubmittedEventArgs args)
{
  string message = $"query: {args.QueryText}";
  if (args.ChosenSuggestion != null)
  {
    message += $" suggestion: {args.ChosenSuggestion}";
  }
  var dlg = new MessageDialog(message);
  await dlg.ShowAsync();
}
```

## 36.10　小结

本章介绍了编写 Windows 应用程序的更多内容，讨论了生命周期与 Windows 桌面应用程序的区别，以及

如何响应 Suspending 事件。

与其他应用程序的交互是使用共享协定实现的。DataTransferManager 用于给其他应用程序提供 HTML 数据。实现“共享目标”协定，就可以接收其他应用程序的数据。

本章展示了已编译绑定的一些特性，比如将属性绑定到方法和分阶段。

下一章继续讨论 XAML 在使用 Xamarin 的 iPhone 和 Android 移动设备上的应用。

# 第 37 章

# Xamarin.Forms

**本章要点**

- Xamarin 开发工具
- Android 和 iOS 的应用程序架构
- 使用 Xamarin.Forms
- 页面
- 导航
- 布局
- 视图
- 数据绑定和命令

**本章源代码下载：**

打开 www.wrox.com 的 Download Code 选项卡可下载本章源代码。源代码也可以在 Xamarin 和 PatternsXamarinShared 目录的 https://github.com/ProfessionalCSharp/ProfessionalCSharp7 中找到。

本章代码分为以下几个主要的示例文件：

- AndroidSample
- iPhoneSample
- BooksAppX (Xamarin.Forms)

## 37.1 Xamarin 开发入门

移动应用开发主要在两个产品之间共享：苹果的 iOS 和谷歌的 Android。iOS 的本机开发是使用编程语言 Objective-C 或 Swift、Cocoa 和 Cocoa Touch 框架完成的。Cocoa 是苹果 API 的名称。在为 Android 开发时，可以使用谷歌的 Android 软件开发工具包，而 Java 是主要的编程语言。

可以使用 C#和 XAML，而不是使用不同的编程语言重写代码。Xamarin 提供跨平台开发，但仍然可以使用本机 API。

由于 Xamarin 被微软收购，并且 Visual Studio 中的 Xamarin 工具集成越来越好，因此许多使用跨平台技术的应用程序可以提高生产率。

本章介绍如何开始创建 Xamarin 应用程序。使用从其他章节学到的有关 C#、.NET Core 和 XAML 的基础，在阅读本章后，就可以使用 Xamarin 开始应用程序开发了。其他专门讨论 Xamarin 的书中还有很多这方面的内容，这是一个好的开始。

**注意：**
*要创建和编译本章中的示例，需要在 Windows 系统上安装 Mobile Development with .NET 工作负载。另一个选项是，可以使用 Visual Studio for Mac。拥有 Android 手机有助于运行 Android 应用程序。要创建和编译 iOS 示例，需要使用 Mac 进行编译，而 iPhone 也很有用。*

### 37.1.1 用 Android 架构 Xamarin

为 Android 手机创建 Xamarin 应用程序时，知道幕后发生了什么是很不错的。图 37-1 给出了一个架构概括。Android 在 Linux 内核上运行。在 Google 中可以看到，Android SDK 在 Android 运行库(ART)之上运行。该图的左侧显示了.NET 部分——使用.NET Mono 运行库的.NET API。Mono 运行库和 ART 在应用程序进程中并行运行。Mono Callable Wrapper(MCW)用于从.NET 中调用 Android SDK。反过来也是可能的：为了从 Android 进入.NET，可使用 Android Callable Wrapper(ACW)。用户过去可能使用过.NET 与 COM(来自 Microsoft 的组件对象模型)。在这里，该体系结构与 Runtime Callable Wrappers(RCW)和 COM Callable Wrappers(CCW)非常相似。为了便于从.NET 进入 Android SDK，Xamarin 创建了 Android 绑定。

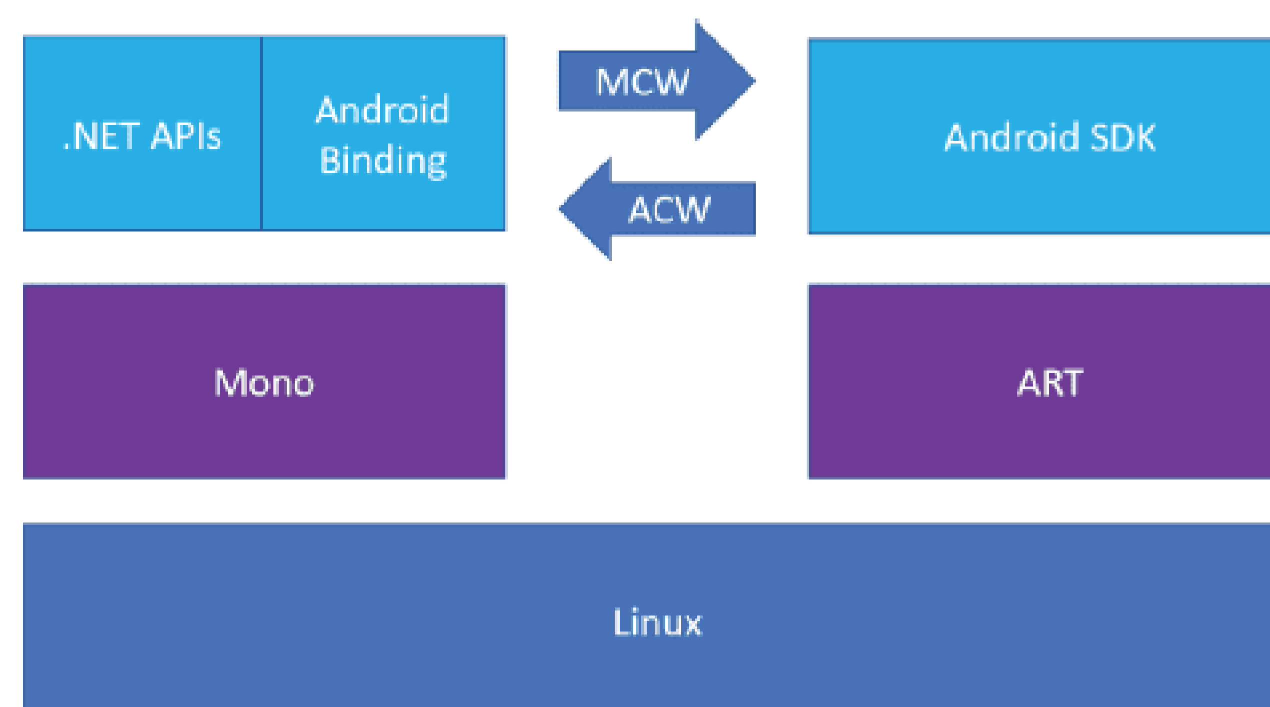


图 37-1

如果想使用的现有 Java 库不属于从.NET 中调用的 Android SDK，则可以创建 MCW。

Xamarin.Android 使 Android 中的 Java API 可用于.NET，而且它非常庞大。这些 API 可以在 https://developer.xamarin.com/api/root/MonoAndroid-lib/上找到。这里只提及几个重要的名称空间：Android.App 名称空间带有 Activity 类和用于长时间运行的后台操作的 Service 类，以及带 UI 元素的 Android.Widget 名称空间。

Xamarin 网站上的文档遗漏了很多可以直接在 https://developer.android.com/reference/packages. html 上阅读的信息。在映射类型时，来自 https://developer.android.com 的信息比较适用。

### 37.1.2 用 iOS 架构 Xamarin

iOS 上 Xamarin 应用程序的体系结构是不同的。由于 iOS 上的安全限制，不允许在设备上执行动态生成的代码。这就是为什么 Xamarin 使用提前(AOT)编译器将从 C#编译器创建的 IL 代码编译为本机代码的原因。这提供了很好的运行性能和更好的启动时间，但它也有一些限制。C#将泛型编译为泛型 IL 类型。使用泛型类型和即时(JIT)编译器，当 IL 代码在运行期间被编译时，会解析泛型。使用 AOT 编译器，IL 代码需要在部署到设备上之前进行编译。所以泛型不能用于某些场景。例如，不能在派生自 NSObject 的类中创建泛型方法。泛型也不能用于 P / Invoke。对于 Android.iOS，P / Invoke 用于使用 C#定义方法，但它使用 Objective-C 中的实现。而且，反射是有限的。不能用反射发射代码，动态生成代码是不允许的。由于 IL 代码已经预编译，Mono iOS Runtime 中的更多功能被禁用，例如元数据验证程序和 JIT 引擎。

图 37-2 给出了 Xamarin 的 iOS 体系结构的概览图。iOS 使用类似于 UNIX 的内核，Objective-C 运行库和 iOS API 在其顶部。.NET API 绑定到 iOS API，以提供相同的功能，并使用 AOT 编译器和删减版的 Mono 运行库。

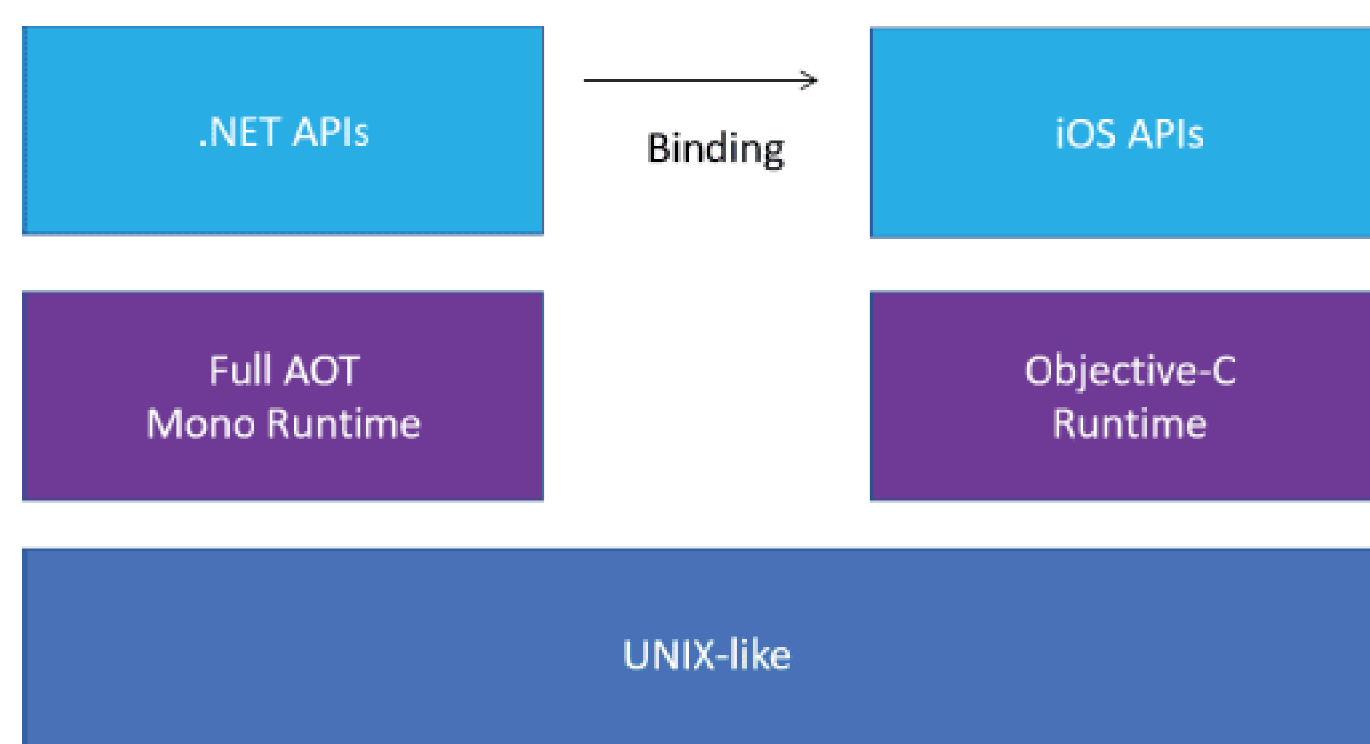



图 37-2

在 https://developer.xamarin.com/api/root/ ios-unified /上可以找到 Xamarin.iOS 的文档。Xamarin for iOS 提供了许多 SDK。 用户界面的主要部分是 UIKit 名称空间。 Apple 的信息可以在 https://developer.apple.com/documentation/uikit 中找到。

幸运的是，可以使用 C#和 XAML 为这些平台开发。 Xamarin 提供跨平台的开发，但它仍然使用平台的所有功能。

**注意：**

为什么用户界面的 iOS 类具有 UI 前缀。 原因是 Objective-C(用于实现本机 iOS 应用程序的编程语言)不支持名称空间。为了避免与其他类的冲突，用户界面的类具有 UI 前缀，而来自其他区域的类具有其他前缀。Xamarin.iOS.Contacts 名称空间中的类具有前缀 CN，例如 CNContact。

**注意：**

Xamarin.iOS(经典)与 Xamarin.iOS 不同，它也标记为“统一”。Xamarin.iOS(经典)仅适用于 iPhone，而更新版本的 Xamarin.iOS 为 iPhone、iPad 和 Mac 使用统一的 API。经典的 API 也仅限于 32 位，不再能在 App Store 中使用。

### 37.1.3　Xamarin.Forms

有了 Xamarin，可以使用 Xamarin.Android 创建 Android 应用程序，使用 Xamarin.iOS 创建 iOS 应用程序，或者使用 Xamarin.Forms 创建适用于 Android、iOS、Windows 和更多平台的应用程序。

有了 Xamarin.Android，就可以使用完整的 API 和 Android SDK 的所有控件。 借助 Xamarin.iOS，可以看到可从 C# 中访问的所有 iOS SDK。 使用 Xamarin.Android 或 Xamarin.iOS 时，可以创建单独的用户界面并使用平台特定的代码，但可以共享业务逻辑和服务。

使用 Xamarin.Forms，可以共享用户界面代码，还可以使用 XAML 代码创建用户界面。 但是，与使用矢量图形绘制元素的通用 Windows 平台相反，Xamarin.Forms 使用每个平台的本机控件呈现用户界面。

使用本机控件具有很大的优势，因为可以获得每个平台上本机控件的外观和性能。 不过，本机控件也有一个缺点。使用 Xamarin.Forms，只能获得可映射到每个平台的控件，得不到仅在 Android 上可用的控件。可以实现一个自定义的特定于平台的渲染器，以使用不能直接从 Xamarin.Forms 上访问的、平台特有的控件或特征。

**注意：**

还记得 Windows 窗体吗？ Windows 窗体是本机 Windows 控件的一个包装器。 也许选择 Xamarin.Forms 这个名字是因为它也是本机控件的包装器。Xamarin.Forms 包装了 iOS、Android 和 Windows 上的本地控件。

## 37.2 Xamarin 开发工具

为 Android 和 iOS 创建应用程序时，需要了解支持所需工具的平台版本。

### 37.2.1 Android

在 Windows 上开发时，我们学习了如何指定开发应用程序的最低版本和目标版本。 这与 Android 非常相似。 对于 Android，用户有 Android 的一个版本以及与编号的 API 级别相对应的代码名称。

用户需要决定想支持平台的哪个版本。支持的原因可能是市场上的分销。表 37-1 列出了 Android 的最新版本及其代码名、API 级别，以及 Android 手机访问 Google Playground 的百分比。 在 2018 年 1 月，Lollipop(5.0)、API 21 级是谷歌支持的最早版本，但 KitKat 4.4 仍占有 12.8%的市场份额。Marshmallow 的市场份额最大，达到 28.6%。可以使用这些信息来决定需要支持的 Android 版本。Oreo 于 2017 年 8 月发布，但其市场份额仍然落后。许多新出售的设备没有安装最新的 Android 版本，并不是所有的手机都可以更新到最新版本。 有关实际发行版，请参阅 https：// developer.android.com/about/dashboards/index.html。

表 37-1

| 版　本 | 代　码　名 | API 级别 | 发　行 | 特　性 |
|---|---|---|---|---|
| 4.4 | KitKat | 19 | 12.8% | 使用云服务打印，新的存储访问框架，基于 NFC 的事务，硬件传感器批处理，步进检测器和步进计数器 |
| 5.0<br>5.1 | Lollipop | 21<br>22 | 25.1% | 材料设计，新部件<br>作业调度器 |
| 6.0 | Marshmallow | 23 | 28.6% | 权限体系结构发生变化，语音交互 |
| 7.0<br>7.1 | Nougat | 24<br>25 | 26.3% | 分屏，增强的通知，<br>应用快捷方式(辅助磁贴) |
| 8.0<br>8.1 | Oreo | 26<br>27 | 0.7% | 自适应图标，可下载的字体，<br>WebView API |

**注意：**

Android 代码名称按其发布时间、按字母顺序排列。 Cupcake(1.5)之后是 Donut(1.6)、Eclair(2.0)、Froyo(2.2)、Gingerbread(2.3)、Honeycomb (3.0)、Ice Cream Sandwich (4.0)、Jelly Bean (4.1)、KitKat (4.4)、Lollipop(5.0)、Marshmallow (6.0)、Nougat (7.0)和 Oreo (8.0)。

为 Android 开发时，需要为支持的版本安装 Android SDK 以及模拟器。 微软最近创建了自己的 Android 和 SDK 工具扩展，以便于安装 Android 平台和工具(见图 37-3)。可以从 Visual Studio Tools | Android | Android SDK Manager 上访问它。

还可以使用通过 Android Emulator Manager 配置的模拟器。如果可以使用真实设备，它就是有益的，它们通常比模拟器更快。 应该在多个设备上测试。来自同一硬件供应商的不同设备可能会有不同的表现。我们可能会从不同的供应商那里购买使用不同平台版本的数百台设备，因此有一个选择。借助 Visual Studio App Center，可以使用 Test Cloud 在数千个物理设备上测试应用程序。只需要创建一个 UI 测试，并在 30 天的免费期后支付每月费用，这可能比购买所有类型的手机便宜。

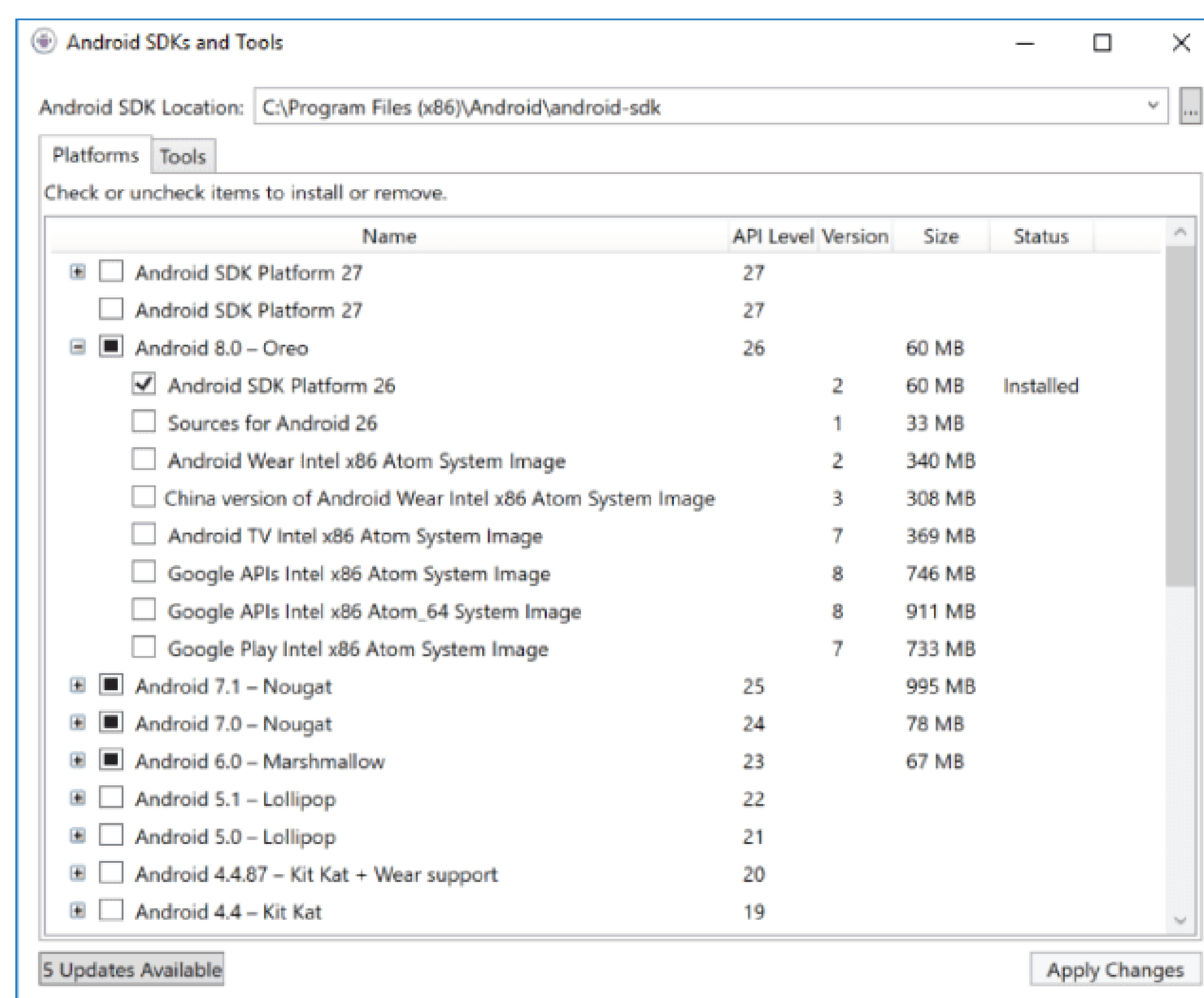



图 37-3

要使用 Android 设备，需要启用它以进行调试。然后将其连接到电脑的 USB 端口。尽管可以通过在 Update 设置中选择 Developer Mode，为开发人员启用 Windows 设备，但使用 Android 需要打开 Settings 页面，并单击构建号七次。

## 37.2.2 iOS

为了开发 iOS 应用程序，需要一台 Mac 来构建 XCode。 如果使用的是 Visual Studio 2016 版本 15.6，则配对变得非常简单。 从 Visual Studio Tools | iOS | Pair to Mac，可以与 Mac 配对，并远程安装所需的 Android SDK。在此过程中，还可以获得注册 Apple 门户的信息，并配置物理设备，以便用于调试。

为 iOS 创建应用程序时，还需要决定支持哪个 iOS 版本。 iOS 的版本有编号，且没有太多需要支持的设备类型。 要查看 iPhone 和 iPad 设备支持哪些 iOS 版本，请访问 http://iossupportmatrix.com/。

与 Android 用户相比，iOS 用户在更新 iOS 版本时速度更快。在 https://developer.apple.com/support/app-store/ 中可以查看使用 App Store 的设备数量。 截至 2017 年 12 月，使用 iOS 11 的用户占 59%，使用 iOS 10 的用户占 33%，使用早期版本的用户仅占 8%。

## 37.2.3 Visual Studio 2017

除了编译 XCode(受 Apple 许可限制)之外，Visual Studio 2017 还具备为 Android、iOS 和 Windows 创建 Xamarin 应用程序所需的全部功能。如果安装了 Mobile Development with .NET 工作负载，就将获得 Android 应用程序、iOS 应用程序和跨平台应用程序的项目模板。Visual Studio 提供了 Android SDK 的安装，可以使用 Android 和 iPhone 模拟器，并且可以为 Android XML(AXML)文件和 iOS 故事板提供设计人员。

## 37.2.4 Visual Studio for Mac

Visual Studio for Mac 起源于 Mac 上用来创建 Xamarin 应用程序的 Xamarin Studio。现在 Visual Studio for Mac 不仅名称改变了，还有更多功能。Visual Studio 的编辑器现在已集成，还可以创建 ASP.NET Core Web 应用程序。

Visual Studio for Mac 包含用于创建 iOS 和 Android 应用程序的项目模板。无法使用 Visual Studio for Mac 创建 Windows 应用程序。 在这里，Visual Studio 需要 Windows 10 系统。

Visual Studio for Mac 与 Visual Studio 一样管理 Android SDK 版本，并为设计人员提供 AXML 文件以及 iOS 故事板。

### 37.2.5 Visual Studio App Center

Visual Studio App Center(https://appcenter.ms)已经在Test Cloud for Android设备中提及。 测试云还支持iPhone和iPad设备。使用UI测试，可以自动在数千个设备上测试应用程序。

测试不是 Visual Studio App Center 的唯一功能。可以创建自动构建程序，并运行单元测试，因为源代码已签入到 Visual Studio Team Services、GitHub 或其他几个代码存储库中。

可以使用 Visual Studio App Center 作为分发工具，将应用分发给 beta 测试人员，并获取有关应用问题的信息。

最后，可以通过应用分析获取生产中的应用的良好报告，了解用户在做什么，发现应用崩溃的位置，自动将问题记录到源代码库中。

> **注意：**
> 第 29 章介绍了 Analytics 和 Visual Studio App Center。

## 37.3 Android 基础

在进入 Xamarin.Forms 之前，最好了解一些平台的基础信息。 第一个应用程序是 Hello Android! App，它是使用 Blank App (Android) 模板创建的。

应该检查的第一个设置是 Android Manifest，可以从 Android Manifest 选项卡的 Project Properties 上访问它(请参见图 37-4)。 在这里，需要定义 Android 的最低版本和目标版本，这与第 33 章中的 Windows 运行库配置类似。这里需要选择应用程序支持的 API 级别。根据想要支持的设备，可以使用可用的功能。

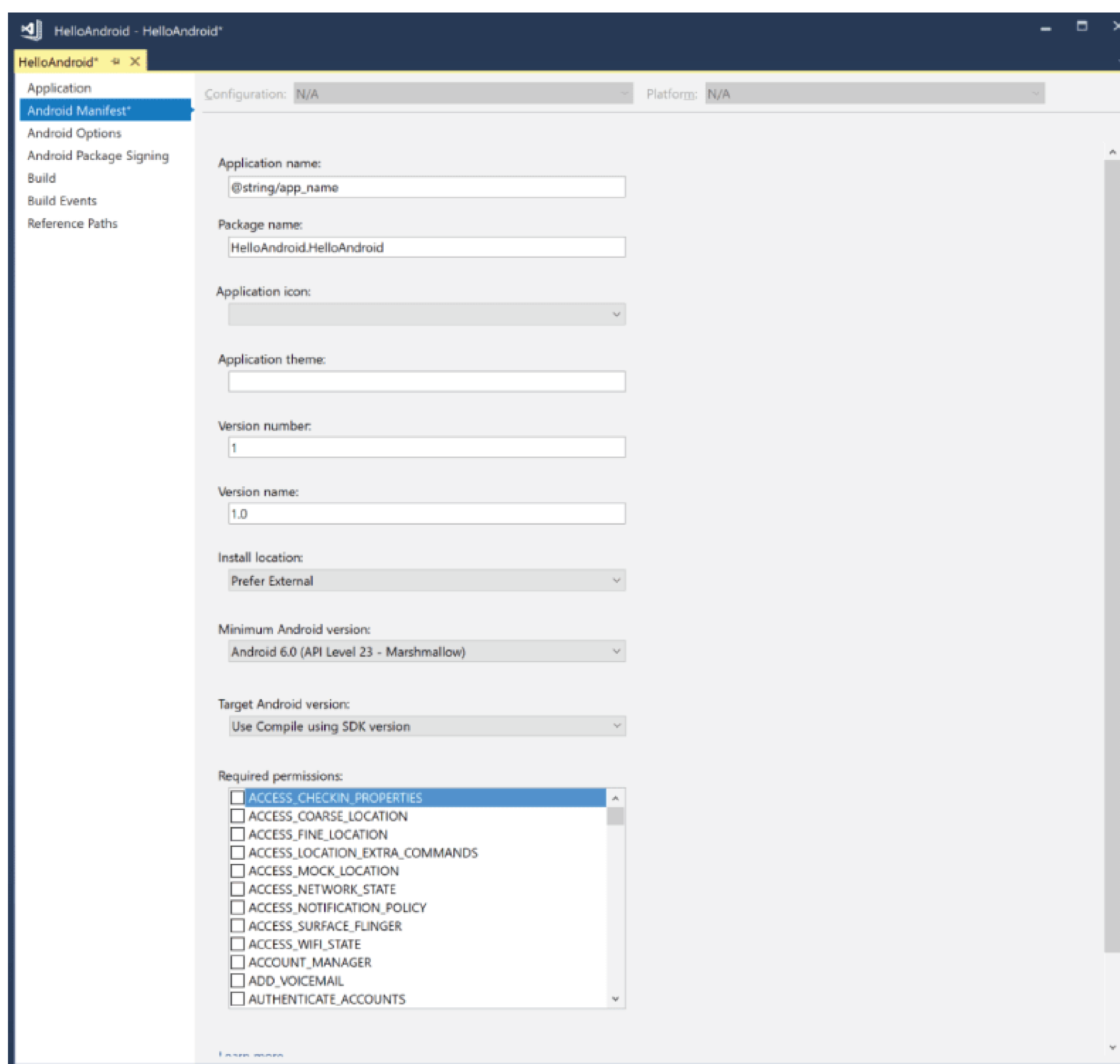



图 37-4

### 37.3.1　活动

使用 Android 时，用户界面屏幕会映射到一个活动上。 为每个视图创建一个活动。 Android 开发利用了 MVC(模型-视图-控制器)模式。活动类是控制器。代码示例显示了 MainActivity——即主视图的活动。每个活动都来自 Activity 类或从 Activity 派生的类。 通过这个类，可以重写由 Activity 基类提供的生命周期方法。 在创建活动时，调用 OnCreate 方法。

调用方法 SetContentView，会定义为此活动定义的用户界面。 UI 在资源中定义，如下一节所述(代码文件 HelloAndroid / MainActivity.cs)：

```
[Activity(Label = "HelloAndroid", MainLauncher = true)]
public class MainActivity : Activity
{
  protected override void OnCreate(Bundle savedInstanceState)
  {
    base.OnCreate(savedInstanceState);
    //...
    // Set our view from the "main" layout resource
    SetContentView(Resource.Layout.Main);
  }
}
```

该活动定义了 Android 应用程序的生命周期。表 37-2 列出了活动的生命周期方法。

表 37-2

| 方　法 | 描　述 |
|---|---|
| OnCreate | 在创建活动时调用此方法，并在用户启动应用程序时执行主要活动。 视图在这个阶段创建。<br>活动的生命周期在 OnCreate 和 OnDestroy 之间定义。<br>这个方法接收一个带参数的 Bundle。 捆绑可用于在活动之间传递数据，并在重新启动后恢复数据。 方法 OnSaveInstanceState 在活动处于后台状态之前调用，因此可以在使用 OnCreate 恢复之前使用此方法写入数据。<br>在此阶段创建视图、初始化变量并将数据绑定到列表 |
| OnStart | 在 OnCreate 之后调用 OnStart。当活动在停止后重新启动时，也会调用此方法。在重新启动时，在 OnStart 之前先调用 OnRestart。<br>活动的可见生命周期是在 OnStart 和 OnStop 之间定义的。<br>在活动可见之前，设置需要的数据 |
| OnResume | OnStart 之后的下一个方法是 OnResume。在 OnResume 之后，活动处于运行状态。<br>活动的前台生命周期在 OnResume 和 OnPause 之间定义。<br>在这里，可以启动动画、显示警报、侦听 GPS 更新、向外部事件添加事件处理程序，这是活动进入前台之前所需的任务 |
| OnPause | 当另一个活动进入前台时，将调用 OnPause。如果用户返回到活动，并且再次出现在前台，那么下一个方法是 OnResume。<br>释放在 OnResume 中分配的资源。该方法的实现应该是快速的，因为只有当 OnPause 方法完成时，用户的下一个活动才会启动 |
| OnStop | 如果活动完全隐藏，则调用 OnStop。如果用户再次导航到该活动，则调用的下一个方法是 OnRestart 和 OnStart。<br>释放在 OnStart 中分配的资源 |
| OnDestroy | 活动被系统破坏，不再可以使用资源。许多应用程序不会重写此方法，因为清理资源已经在 OnStop 和 OnPause 中完成了。<br>如果有一个长时间运行的资源，比如在 OnCreate 中启动的后台线程，就应该在这里重新设置状态。<br>这个方法不能保证被调用。活动可以被用户或操作系统停止，而不调用 OnDestroy |

### 37.3.2 资源

在 Solution Explorer 中，可以看到 Resources 文件夹带有多个子文件夹。drawable 文件夹用于存储图像。视图在 layout 文件夹中定义，values 文件夹包含简单的字符串。

在示例代码中，字符串资源被更新，以包含用于按钮文本、应用程序名称和由 hello 标识的字符串(代码文件 HelloAndroid/Resources/values/Strings.xml)。

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="main_button1_text">Click Me!</string>
  <string name="app_name">Hello Android</string>
  <string name="hello">Hello Android!</string>
  <string name="somedata_clickeditem_title">Clicked Item</string>
</resources>
```

视图是使用 AXML 文件定义的，它与 XAML 文件有相似之处。LinearLayout 是一种安排子元素的布局控件。可以将 LinearLayout 与 StackPanel 类进行比较，它具有相同的功能。在示例应用程序中，在布局内部添加一个按钮控件。可以将按钮控件从工具箱拖动到设计器。带有属性的 android 前缀是名称空间 http://schemas.android.com/apk/res/android 的别名。对于按钮，text 设置为字符串资源中的资源字符串 main_button_text，宽度设置为与父节点的宽度相匹配，高度设置为换行，以免一行放不下。将 id 分配给 @+id/button1，会创建一个新的 ID，名称 button1.@+id 是在设计器生成的文件 Resource.Designer.cs 中创建新 ID 来访问按钮的快捷方式(代码文件 HelloAndroid/ Resources/layout/Main.axml)：

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:minWidth="25px"
  android:minHeight="25px">
  <Button
    android:text="@string/main_button_text"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:id="@+id/button1" />
</LinearLayout>
```

在活动中，可以使用基类 Activity 中的 FindFyById 方法访问按钮控件，通过 Resource.Id.button1 引用传递的 ID。在代码片段中，检索按钮，以添加单击事件处理程序。处理程序的实现代码向用户显示一个通知(代码文件 HelloAndroid / MainActivity.cs)：

```
protected override void OnCreate(Bundle savedInstanceState)
{
  base.OnCreate(savedInstanceState);

  SetContentView(Resource.Layout.Main);

  Button button1 = FindViewById<Button>(Resource.Id.button1);
  button1.Click += (sender, e) =>
    Toast.MakeText(ApplicationContext, "Hello Android!",
      ToastLength.Long).Show();
}
```

在模拟器或物理设备上运行应用程序时，按钮会显示，可以单击它，会看到一个 toast，如图 37-5 所示。

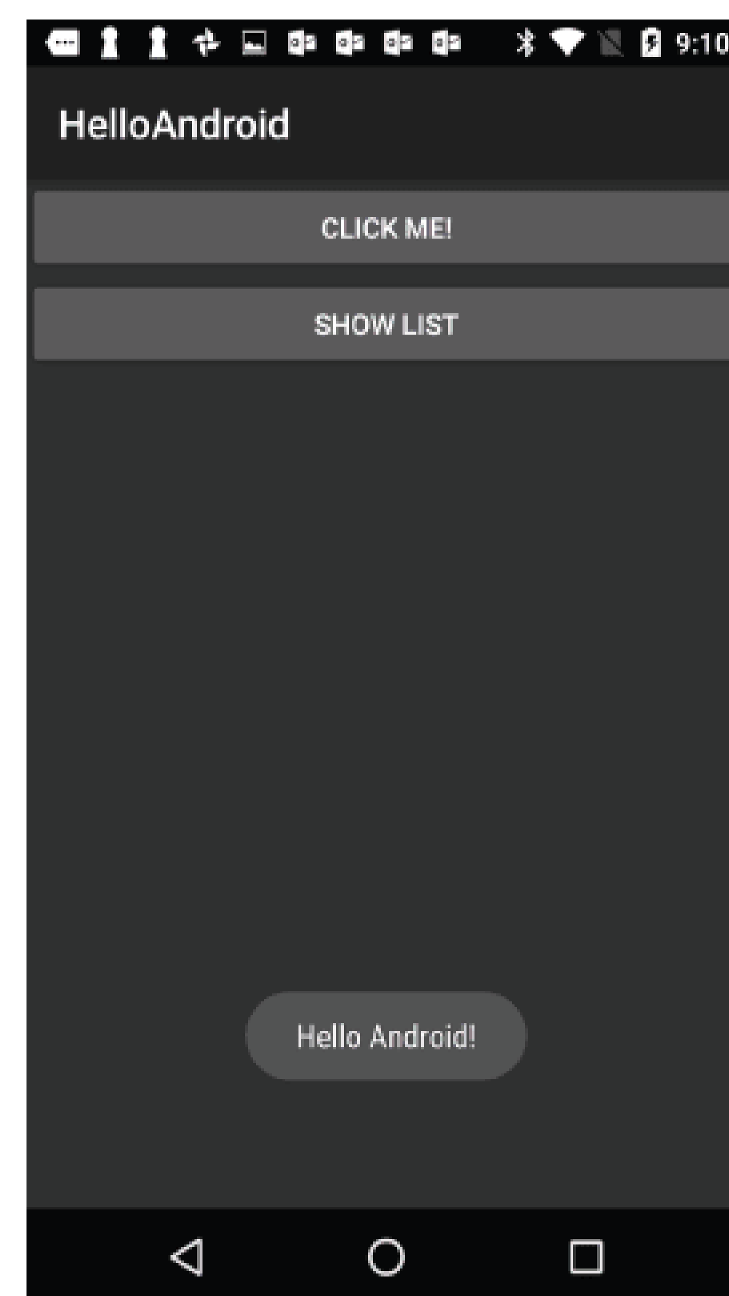


图 37-5

### 37.3.3 显示列表

一个常见的场景是导航到页面，并显示列表。这是通过应用程序的下一个增强来显示的。为此，先创建一个模型。

### 1. 定义模型

该模型是一个简单的类,定义了一些属性并重写了 ToString 方法(代码文件 HelloAndroid/Models/SomeData.cs)。

```
public class SomeData
{
  public int Number { get; set; }
  public string Text { get; set; }
  public override string ToString() => $"{Number} {Text}";
}
```

### 2. 使用 ListActivity

接下来，创建一个活动。现在使用 ListActivity 基类，而不是从 Activity 基类中派生活动。ListActivity 定义了用于在列表中选择项的虚拟属性和方法。在创建 SomeDataListActivity 时，将创建一个包含示例数据的集合，并将其存储在_items 字段中。在 OnCreate 方法中，基类的 ListAdapter 属性也分配给 SomeDataListAdapter 的一个新实例。在 MainActivity 中，活动的用户界面由使用 SetContentView 的资源定义。这里，使用适配器来提供用户界面。该活动还定义了重写方法 OnListItemClick，以显示所选项(代码文件 HelloAndroid/ SomeDataListActivity.cs)：

```
[Activity(Label = "SomeDataListActivity")]
public class SomeDataListActivity : ListActivity
{
  private IList<SomeData> _items;
  protected override void OnCreate(Bundle savedInstanceState)
  {
    base.OnCreate(savedInstanceState);

    _items = Enumerable.Range(0, 100)
      .Select(i => new SomeData { Number = i, Text = $"sample {i}" })
      .ToList();
    ListAdapter = new SomeDataListAdapter(this, _items);
  }

  protected override void OnListItemClick(ListView l, View v, int position,
    long id)
  {
    //... implementation of the OnListItemClick event handler
  }
}
```

### 3. 实现一个适配器

SomeDataListAdapter 用于为 SomeDataListActivity 定义 UI。这个类的核心是重写方法 GetView，它需要返回列表中每一项的视图。该方法接收父 ViewGroup，其中将显示该项的视图。LayoutInflater 将 AXML 文件放大到一个视图中。可以为资源中的项创建视图。代码示例为 Android 使用一个预定义的 AXML 文件。Android.Resource.Layout.SimpleListItem1 返回预定义 AXML 文件的 ID。这个布局使用 TextView 元素定义一个简单的项。TextView 元素使用 FindViewById 来访问，其中 Text 属性分配给项中的信息 (代码文件 HelloAndroid/SomeDataListAdapter.cs)：

```
public class SomeDataListAdapter : BaseAdapter
{
  private readonly Activity _activity;
  private readonly IList<SomeData> _items;

  public SomeDataListAdapter(Activity activity, IList<SomeData> items)
  {
    _activity = activity;
    _items = items;
  }

  public override Java.Lang.Object GetItem(int position) => position;

  public override long GetItemId(int position) => position;

  public override View GetView(int position, View convertView,
    ViewGroup parent)
  {
```

```
        var view = convertView;
        if (view == null)
        {
          view = _activity.LayoutInflater
            .Inflate(Android.Resource.Layout.SimpleListItem1, null);
        }
        view.FindViewById<TextView>(Android.Resource.Id.Text1).Text =
          $"{_items[position].Number}: {_items[position].Text}";
        return view;
      }

      public override int Count => _items.Count;
    }
```

**注意：**

要了解 SimpleListItem1 和其他预定义布局是如何定义的，请参阅 https://github.com/aosp-mirror/platform_frameworks_base/tree/master/core/res/res/layout。

#### 4. 用 Android 导航

要导航到 SomeDataList 活动，对于 Android，需要启动活动，并传递活动类型。这是在 MainActivity 的 showlistButton 中完成的(代码文件 HelloAndroid/ MainActivity.cs)：

```
showlistButton.Click += (sender, e) =>
  StartActivity(typeof(SomeDataListActivity));
```

运行该应用程序时，可以看到从 SomeDataListAdapter 返回的条目列表，其中包含预定义的 SimpleListItem1，如图 37-6 所示。




图 37-6

### 37.3.4　显示消息

在 Android 应用程序的第一个按钮上，在单击该按钮之后，就会显示 Toast。如何从 UWP 中显示像 MessageDialog 这样的东西？对于 Android，可以使用 AlertDialog。单击列表中的项时，会使用这个对话框。

在使用对话框之前，需要创建它。要创建 AlertDialog，需要实例化内部 Builder 类。使用 AlertDialog.Builder，可以使用 SetMessage 定义要显示的消息，并使用 SetTitle 来定义要显示的标题。为了定义应显示的按钮，Builder 类定义了一些方法，例如 SetNeutralButton、SetPositiveButton 和 SetNegativeButton。甚至可以创建多选项 SetMultiChoiceItems。在代码片段中，SetNeutralButton 是使用预定义的资源字符串 Android.Resource.String.Ok 来设置的。在配置构建器之后，AlertDialog 将从 Builder 的 Create 方法中创建。最后，需要调用 Show 方法来显示对话框(代码文件 HelloAndroid/SomeDataListActivity.cs)：

```
    protected override void OnListItemClick(ListView l, View v, int
position,
      long id)
    {
      AlertDialog.Builder builder = new AlertDialog.Builder(this);
      builder.SetMessage($"clicked {_items[position]}")
        .SetTitle(Resource.String.somedata_clickeditem_title);

      builder.SetNeutralButton(Android.Resource.String.Ok, (sender, e) =>
      {
        // user clicked the ok button
      });
      AlertDialog dialog = builder.Create();

      dialog.Show();
    }
```

运行该应用程序，AlertDialog 如图 37-7 所示。

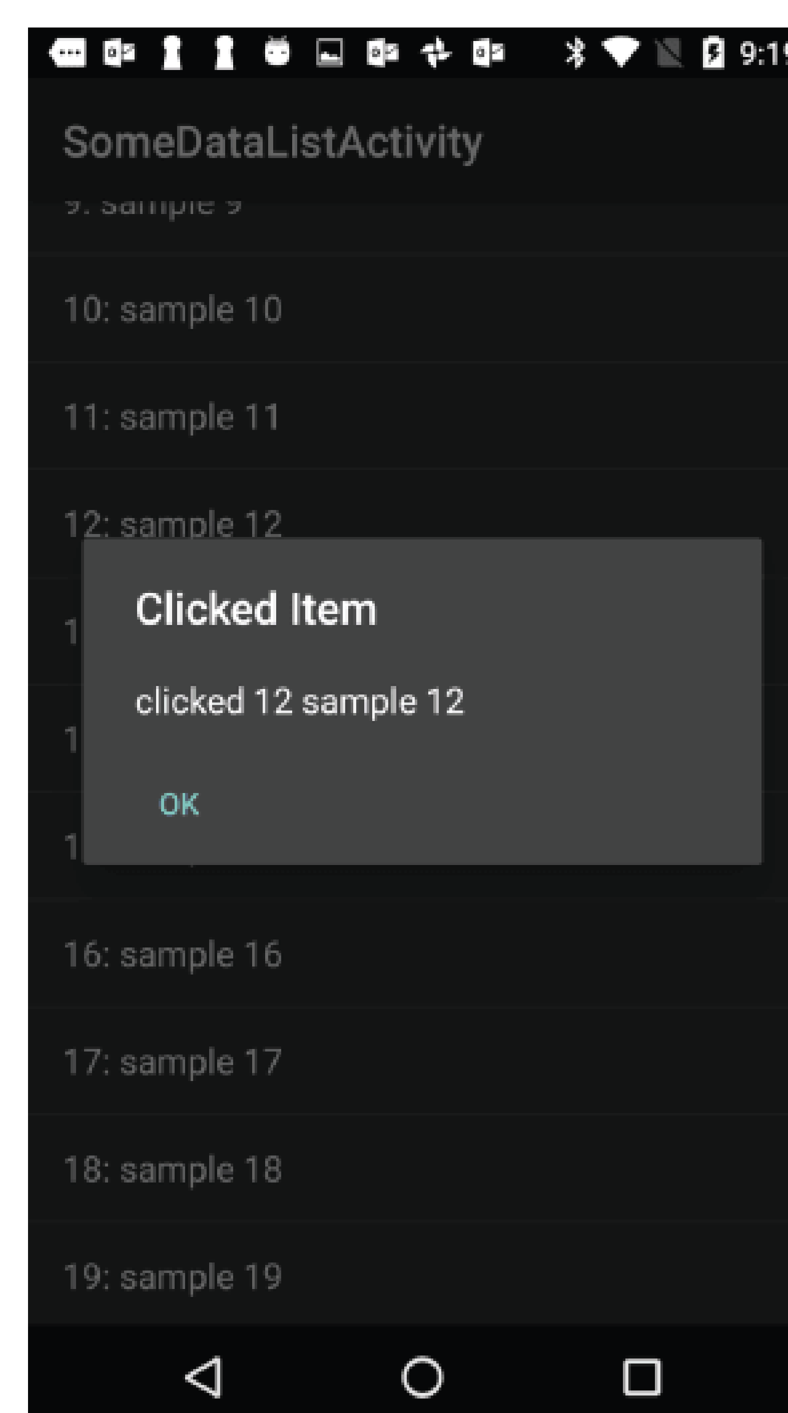



图 37-7

用 Android 构建应用程序提供了一些有趣的方面，但其开发与创建

Windows 应用程序非常不同。与 iPhone 相似的场景是怎样的呢?

## 37.4 iOS 基础

使用 Android.iOS，创建 HelloiOS 应用程序。要创建和构建这个应用程序，需要一个安装了 Xcode 的 Mac，这样构建过程就可以使用 Xcode 引擎了。所创建的应用程序是从使用 Single View App (iPhone)项目模板开始的。

所生成的文件 Info.plist 包含的信息与 Windows 应用程序中的 package.manifest 文件类似(参见图 37-8)。在这里，可以配置部署目标，以支持在应用程序中支持的 iOS 版本，为图标添加可视资产，配置功能(例如，当需要后台功能时)，并配置用于启动应用程序的文档类型。

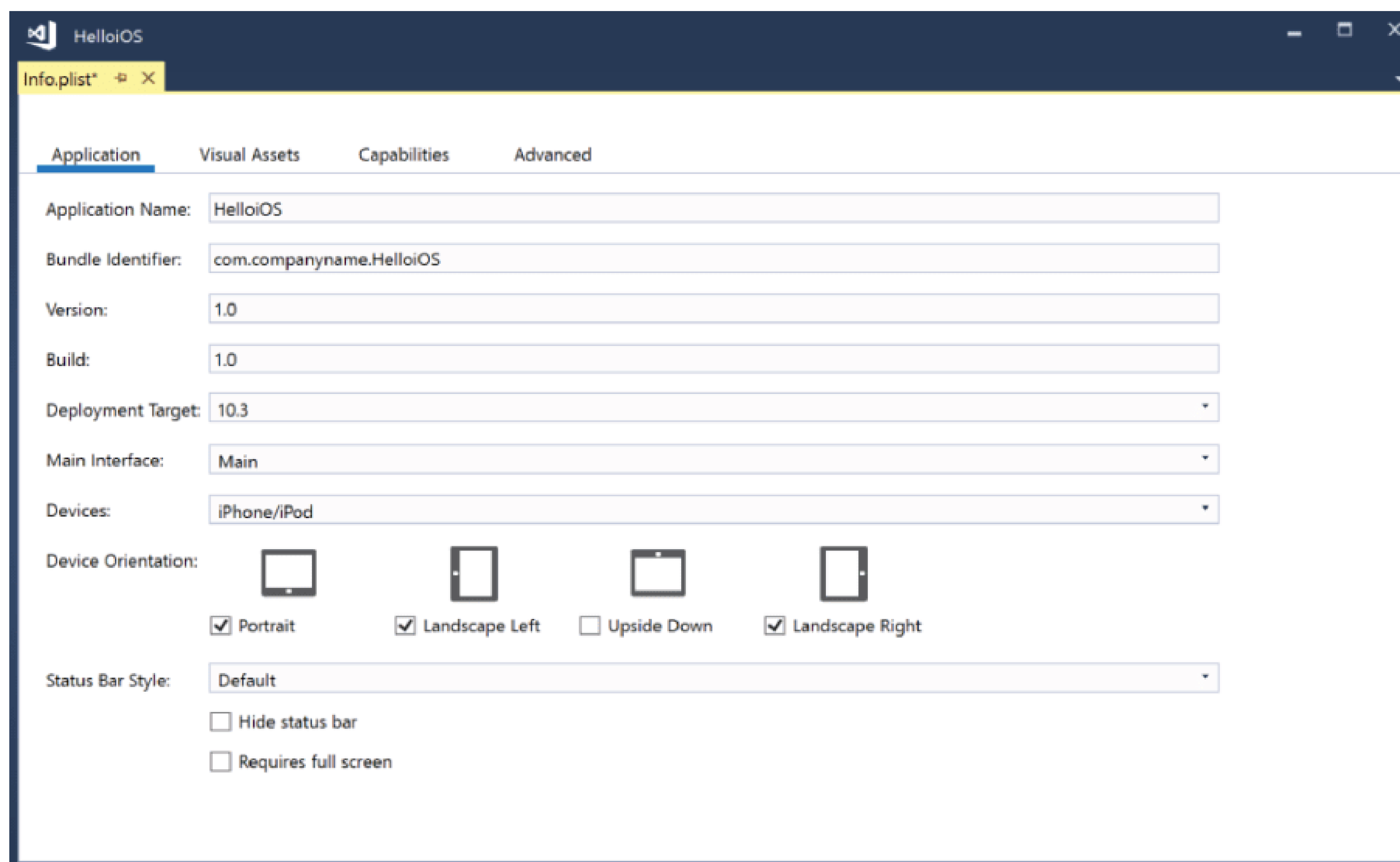



图 37-8

### 37.4.1 iOS 应用程序结构

在生成的代码中，有一个 Main()方法作为应用程序的入口点。调用 UIApplication 类的静态 Main()方法，消息就开始对事件做出反应。这需要处理事件的应用程序委托的名称(代码文件 HelloiOS/Main.cs)：

```
public class Application
{
  static void Main(string[] args)
  {
    UIApplication.Main(args, null, "AppDelegate");
  }
}
```

AppDelegate 类定义了应用程序的生命周期。它实现的功能可以在应用程序启动时用于初始化(FinishedLaunching)，在如下情况下通知应用程序：使用 OnResignActivation 从活动状态转换到不活动状态时 (例如，打一个电话)；当用户将另一个应用程序移动到前台(DidEnterBackground)来保存用户数据时；当应用程序回到前台(WillEnterForeground) 时；当应用程序被激活(OnActivated)，以重启暂停任务并刷新用户界面时；当应用程序被终止(WillTerminate)，以保存数据时(代码文件 HelloiOS/AppDelegate.cs)：

```
[Register("AppDelegate")]
public class AppDelegate : UIApplicationDelegate
{
  public override UIWindow Window
  {
    get;
```

```
    set;
  }

  public override bool FinishedLaunching(UIApplication application,
    NSDictionary launchOptions)
  {
    return true;
  }

  public override void OnResignActivation(UIApplication application)
  {
  }

  public override void DidEnterBackground(UIApplication application)
  {
  }

  public override void WillEnterForeground(UIApplication application)
  {
  }

  public override void OnActivated(UIApplication application)
  {
  }

  public override void WillTerminate(UIApplication application)
  {
  }
}
```

## 37.4.2　故事板

iOS 应用程序也基于 MVC 模式。然而，在 iOS 中，用户界面是围绕故事板来组织的。故事板定义了用户与视图交互的流程。通过故事板设计器(参见图 37-9)，可以使用工具箱将控制器和控件添加到故事板中(参见图 37-10)。

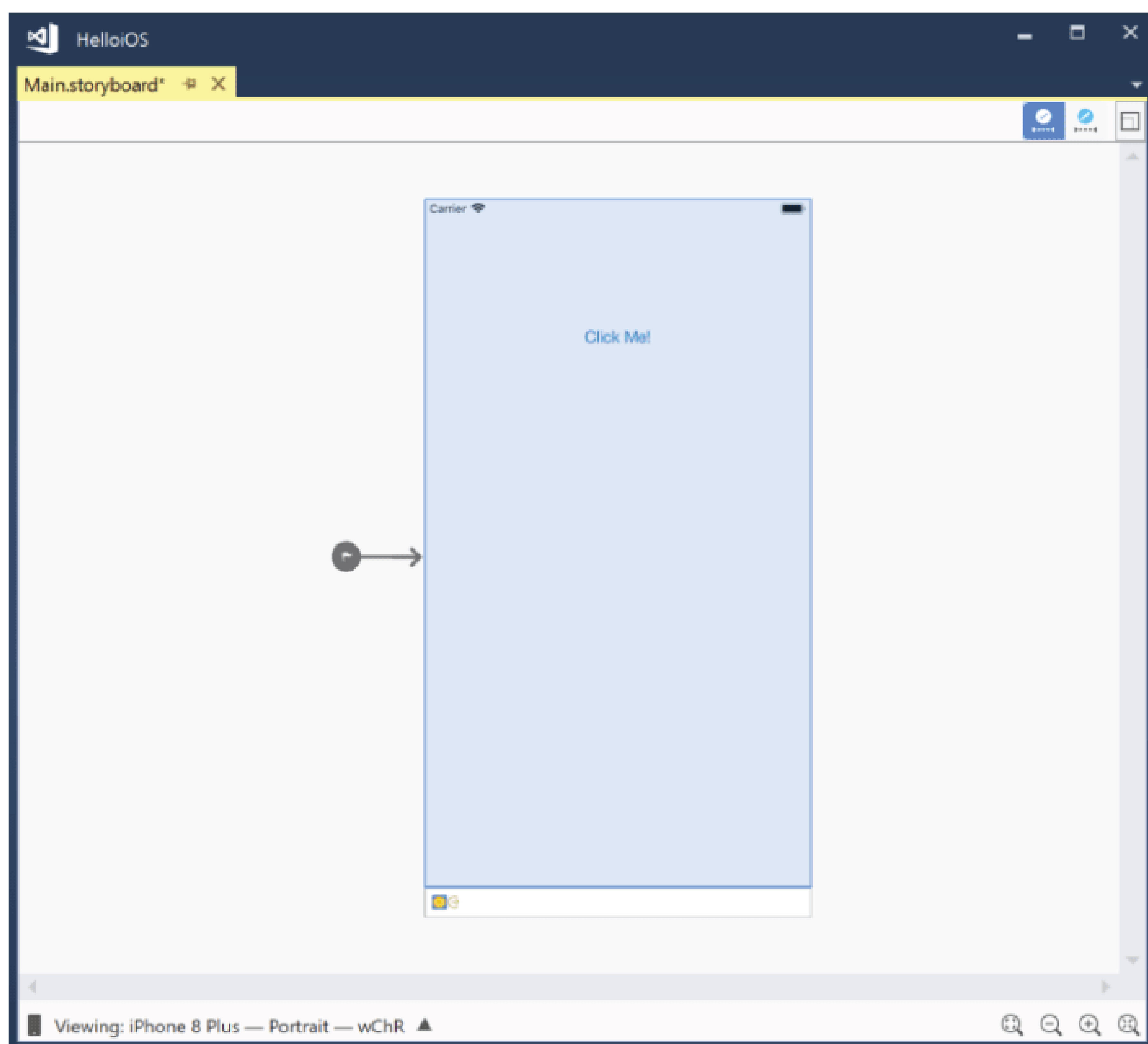



图 37-9

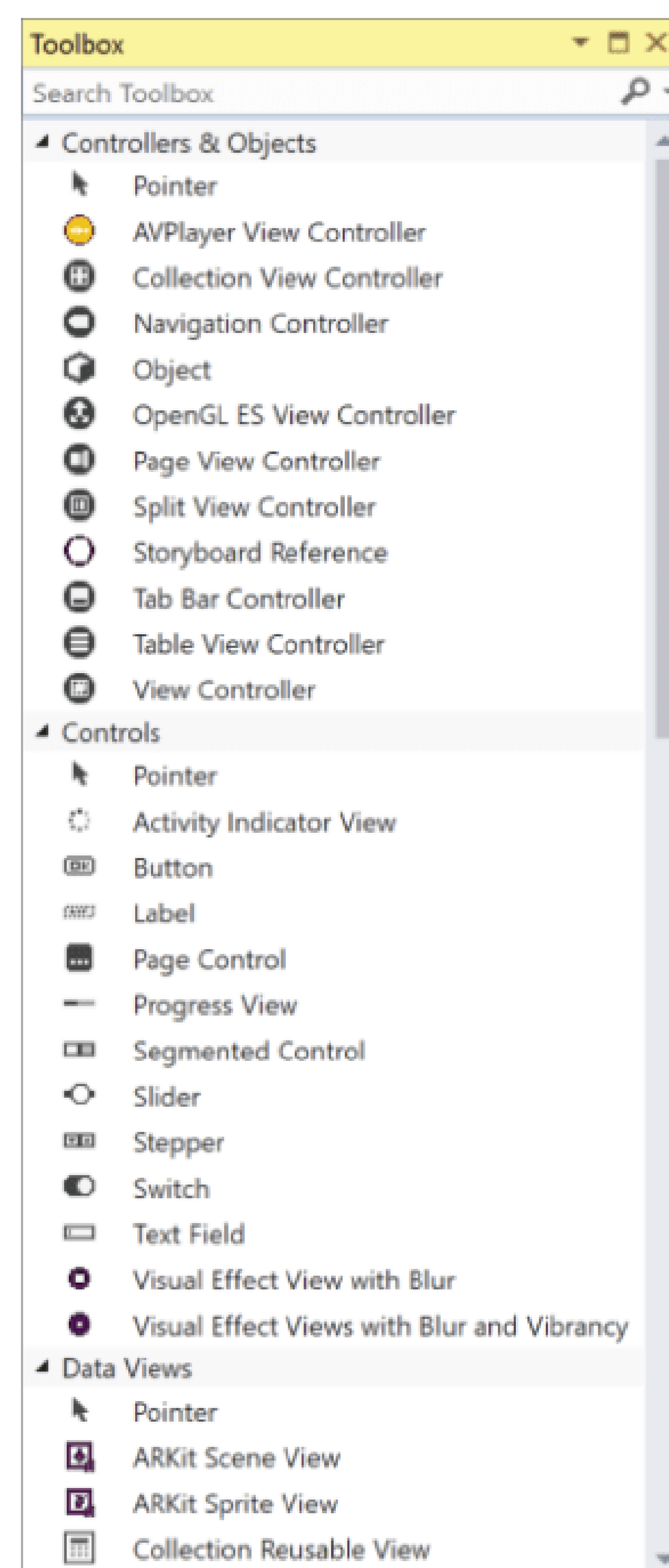



图 37-10

在视图中添加控件(如按钮)时，可以指定属性(参见图 37-11)、布局和事件(参见图 37-12)。可以使用 Touch Down 或 Touch Up Inside，而不是将 Click 事件赋予按钮。当按钮内的触摸事件结束时，会触发 Touch Up Inside。当按钮外的触摸事件(例如，用户将手指移到按钮的外部)时，会触发 Touch Up Outside。Touch Down 在 Touch In 事件发生前触发，并且总是在触摸时被触发。对于典型的单击事件，请使用 Touch Up Inside。

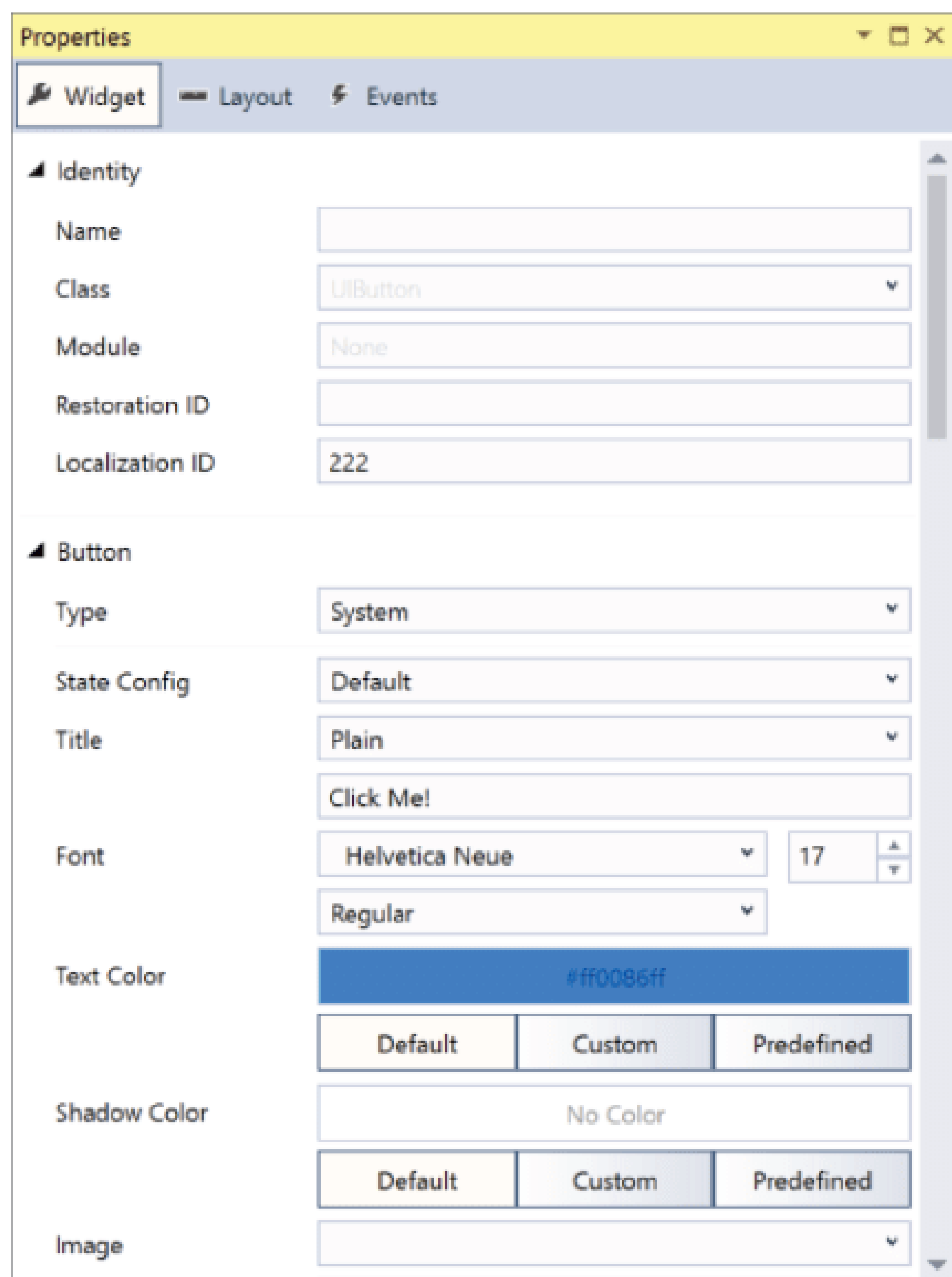



图 37-11

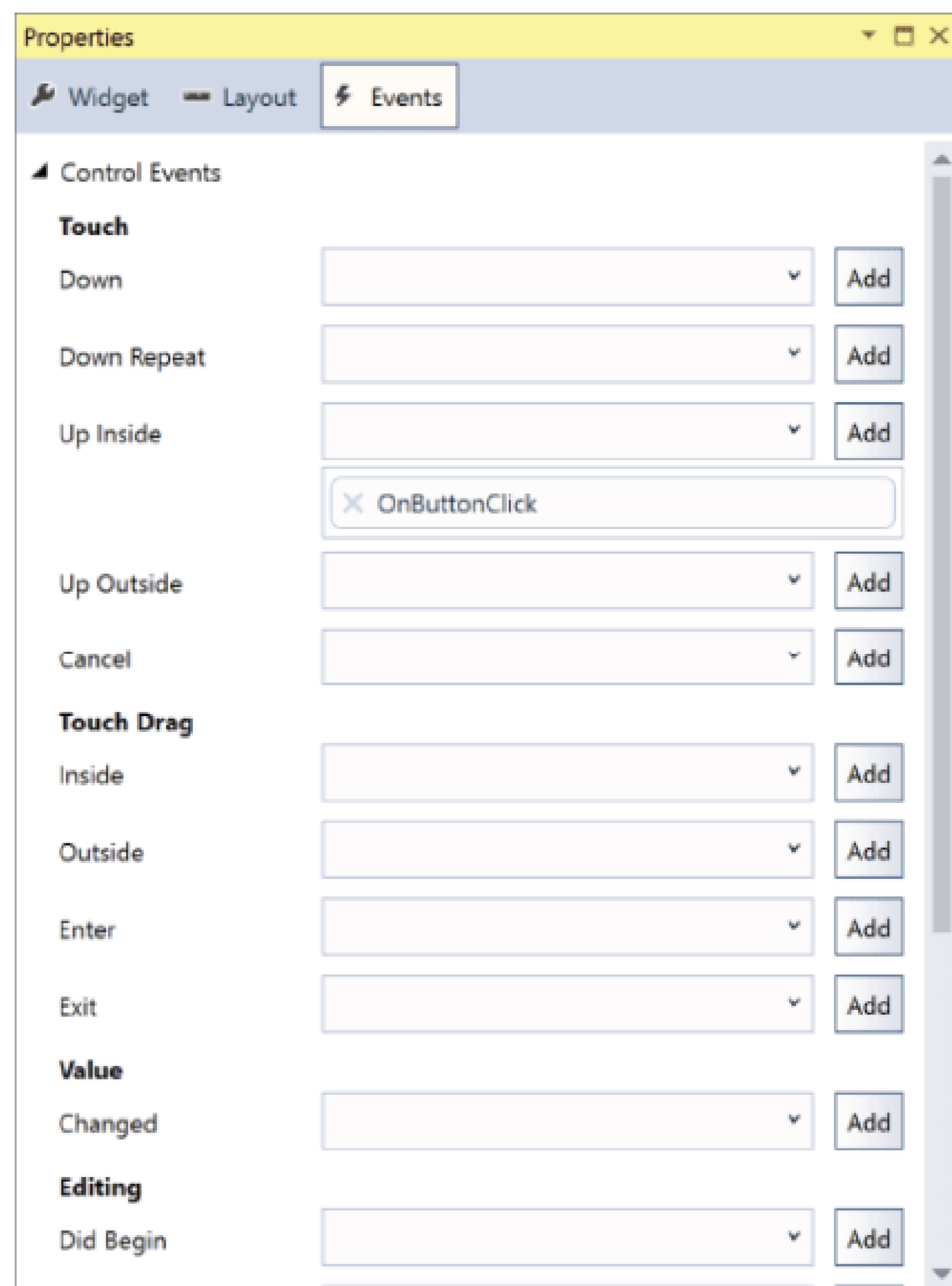



图 37-12

在故事板设计器中，可以通过单击视图左下角的图标来访问与视图关联的控制器，这会显示关联控制器的属性(参见图 37-13)。选择 ViewController 时，可以看到启用了 Is Initial View Controller 设置。该设置使这个视图成为初始 UI。

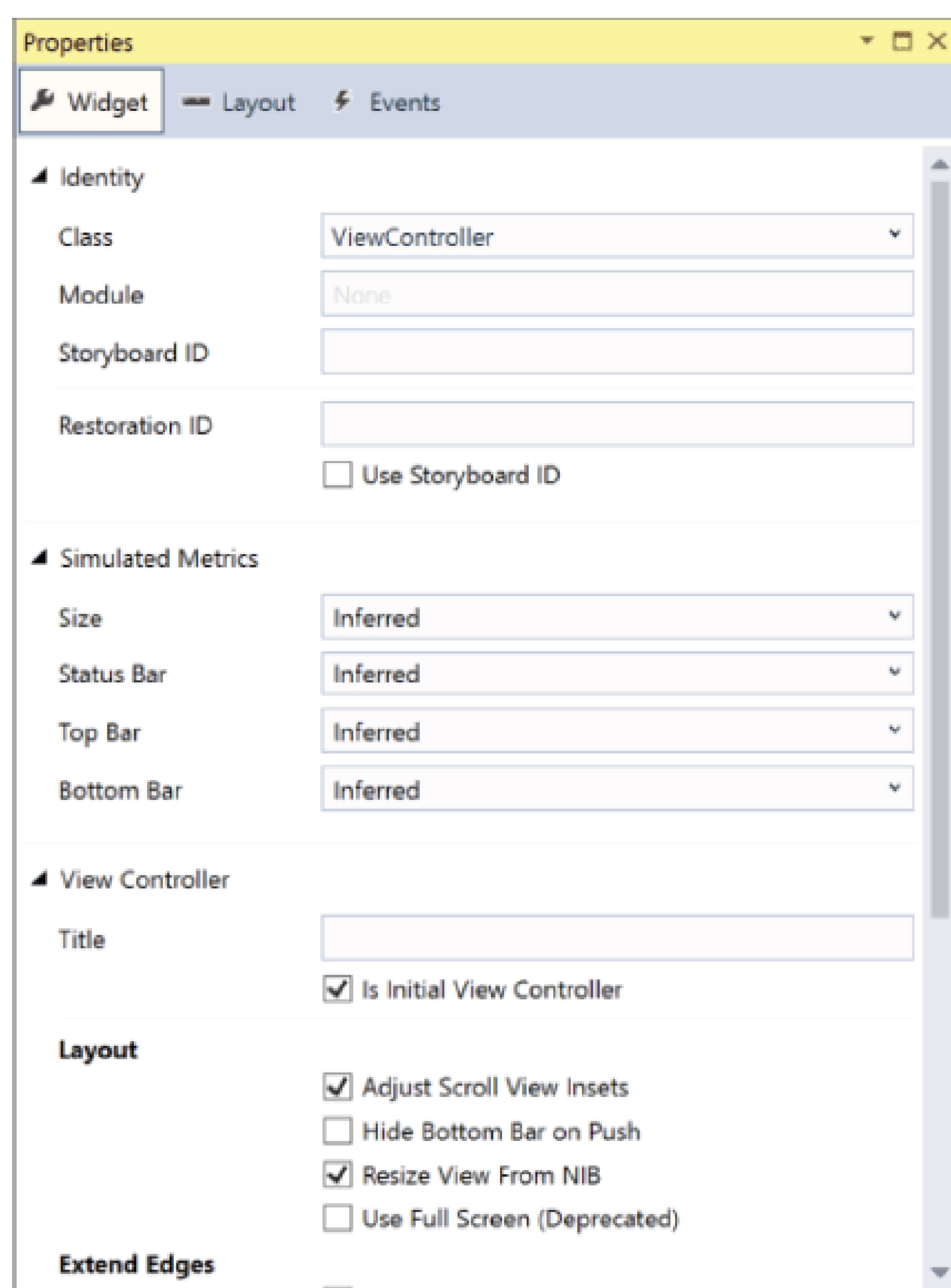



图 37-13

故事板设计器创建了一个 XML 文件，其中 Button 由按钮元素定义，单击事件用一个连接和一个操作定义。使用该操作定义的选择器在关联控制器的设计器生成的代码文件中定义了一个部分方法：

```
<connections>
  <action selector="OnButtonClick:" destination="BYZ-38-t0r" id="1110"
    eventType="touchUpInside"/>
</connections>
```

### 37.4.3 控制器

与故事板视图相关的控制器类派生自基类 UIViewController。控制器管理着应用程序模型与视图之间的交互。

基类 UIViewController 管理视图的布局，包括大小调整和对方向变化的响应。在这个文件中，还实现了处理程序，以响应来自 UI 元素的事件，如下所示(代码文件 HelloiOS/ViewController.cs)：

```
public partial class ViewController : UIViewController
{
  public ViewController(IntPtr handle) : base(handle)
  {
  }

  public override void ViewDidLoad()
  {
    base.ViewDidLoad();
  }

  public override void DidReceiveMemoryWarning()
  {
    base.DidReceiveMemoryWarning();
  }
  //...
}
```

### 37.4.4 显示消息

通过故事板中的按钮声明，为 touchUpInside 事件定义 OnButtonClick。这在控制器的设计器生成的文件中定义了一个名为 OnButtonClick 的局部方法的声明。Action 属性将其映射到事件上(代码文件 HelloiOS/ViewController.designer.cs)：

```
[Action ("OnButtonClick:")]
[GeneratedCode ("iOS Designer", "1.0")]
partial void OnButtonClick (UIKit.UIButton sender);
```

在自定义控制器实现类中，现在可以实现 OnButtonClick 方法了。在样例应用程序中，创建了一个新的 UIAlertView，其中设置了 Title 和 Message 属性。使用 AddButton 方法添加了一个 Close 按钮。最后，要显示对话框，应调用 Show 方法(代码文件 HelloiOS/ViewController.cs)：

```
public partial class ViewController : UIViewController
{
  //...
  partial void OnButtonClick(UIButton sender)
  {
    var alert = new UIAlertView
    {
      Title = "Hello",
      Message = "Hello iOS!",
    };
    alert.AddButton("Close");
    alert.Clicked += (sender1, e) =>
    {
      // dialog closed
    };
    alert.Show();
  }
}
```

在模拟器中运行应用程序时，可以看到带有 UIAltertView 的按钮，如图 37-14 所示。

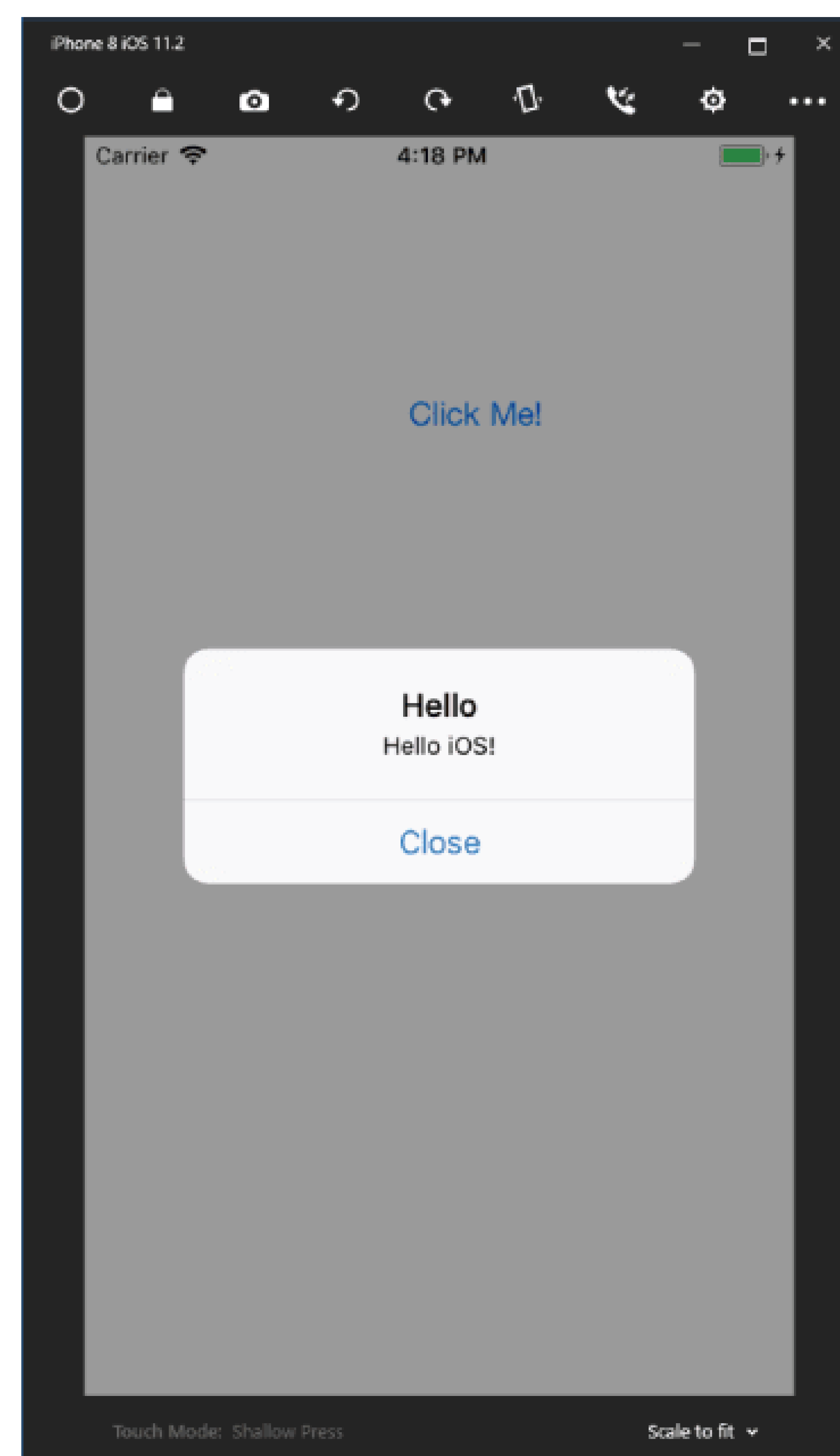


图 37-14

## 37.5 Xamarin. Forms 应用程序

现在了解了关于 Android 和 iOS 的一些原则，下面从 Xamarin.Forms 开始。这个项目命名为 BooksAppX，并使用第 34 章中用于视图模型和核心功能的库。这些库的项目位于文件夹 PatternsXamarinShared 中。

第 34 章中的 Windows 应用程序项目展示了书籍列表和一本书的详细信息。同样的功能现在将使用 Xamarin.Forms 来实现——使用相同的库。

要创建 Xamarin.Forms 项目，模板 Cross-Platform App (Xamarin.Forms)是一个很好的开端。使用这个项目模板，在 Windows 上可以选择三个平台、UI 技术和代码共享策略，如图 37-15 所示。

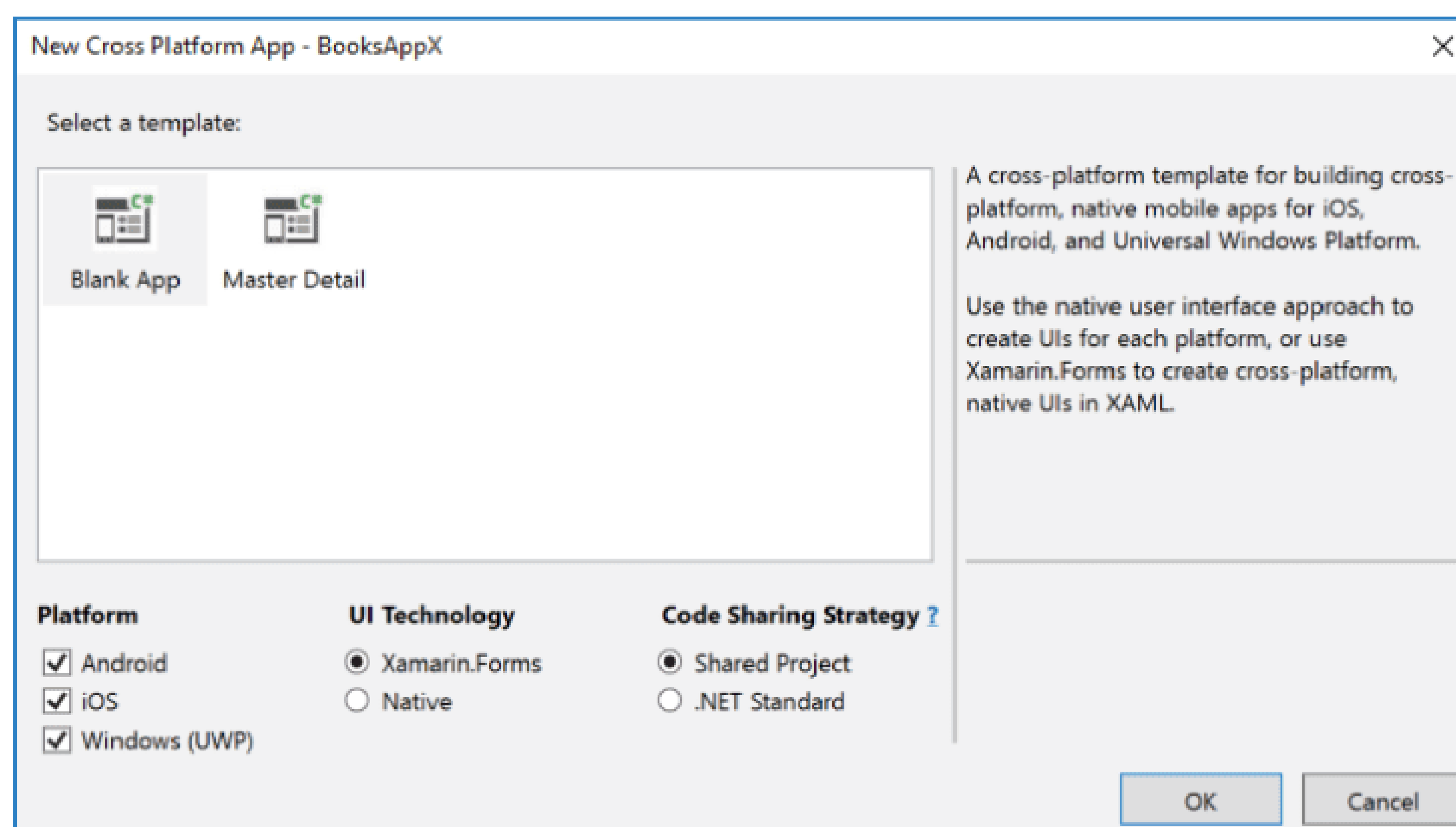



图 37-15

使用 Xamarin.Forms 可以为 Android、iOS 和 Windows 平台创建跨平台的应用程序。在撰写本文时，Xamarin.Forms for WPF 的预览已经是可用的，所以 Xamarin.Forms 不会在这三个平台上停止。要选择的 UI 技术是 Xamarin.Forms 或 Native。选择 Native 时，就会创建本地应用程序。用户界面在此场景中不共享。但是，可以使用公共的视图-模型和公共服务。通过 Xamarin.Forms，使用 XAML 为所有这些平台创建用户界面。这是本章从现在开始使用的选项。使用代码共享策略，可以在共享项目和.NET 标准之间选择。使用.NET 标准，创建一个包含 UI 代码的库，在所选的平台之间共享。选择共享项目时，源代码会构建到所选的每个平台应用程序中。库不共享，但是源代码是共享的。在示例项目中，在模式和 Xamarin 章节共享的两个项目将共享一个.NET 标准库，但是 XAML 代码与共享项目共享。

> **注意：**
> *有关共享项目和.NET 标准的更多信息，请阅读第 19 章。*

使用新的跨平台应用程序时，可以选择的模板是 Blank App 和 Master Detail 应用程序，虽然这款应用程序会有一些主从功能，但我们还是先从空白开始，并在前面创建的库的帮助下实现主从功能。

创建项目 BooksAppX 时，创建了四个项目：用于通用 Windows 平台、iOS 和 Android 的三个托管项目，以及一个包含 XAML 文件的共享项目。由于前面已经使用这三个平台创建了项目，因此托管代码将非常熟悉。下面通过这三种技术来完成启动。

### 37.5.1 托管 Xamarin 的 Windows 应用程序

生成的 App 类非常类似于第 33 章中用来创建 Windows 应用程序的 App 类。不同的是 Xamarin.Forms 现在在重写方法 OnLaunched 中初始化(代码文件 BooksAppX/BooksAppX/UWP/App.xaml.cs)：

```
protected override void OnLaunched(LaunchActivatedEventArgs e)
{
```

```
  Frame rootFrame = Window.Current.Content as Frame;
  if (rootFrame == null)
  {
    rootFrame = new Frame();

    rootFrame.NavigationFailed += OnNavigationFailed;

    Xamarin.Forms.Forms.Init(e);

    if (e.PreviousExecutionState == ApplicationExecutionState.Terminated)
    {
      //TODO: Load state from previously suspended application
    }

    Window.Current.Content = rootFrame;
  }

  if (rootFrame.Content == null)
  {
    rootFrame.Navigate(typeof(MainPage), e.Arguments);
  }
  Window.Current.Activate();
}
```

现在，从 OnLaunched 方法中创建的 MainPage 是一个派生自 Xamarin.Forms.Platform.UWP.WindowsPage 的类。这个类驻留 Xamarin.Forms 页面。公共代码在 MainPage 类的构造函数中加载。LoadApplication 是 Xamarin.Forms 的开始，它从共享项目中创建一个新的 App 类，并将实例传递给 LoadApplication 方法(代码文件 BooksAppX/BooksAppX/UWP/MainPage.xaml.cs)：

```
public MainPage()
{
  this.InitializeComponent();

  LoadApplication(new BooksAppX.App());
}
```

**注意：**

不要被 UWP 场景中的两个 MainPage 和两个 App 类迷惑。一个 App 和 MainPage 类在 UWP 项目中。这些类仅由 Windows 应用程序使用，其他 App 和 MainPage 类都在共享项目中。这些类由 UWP、Android 和 iOS 应用程序使用。这些类的名称空间是不同的。

### 37.5.2 托管 Xamarin 的 Android

在 Android 项目中，会发现 MainActivity。这个活动现在派生自基类 Xamarin.Forms.Platform.Android.FormsAppCompatActivity。Xamarin.Forms 的初始化和公共 App 类的实例化在 OnCreate 方法中(代码文件 BooksAppX/BooksAppX. android /MainActivity.cs)：

```
[Activity(Label = "BooksAppX", Icon = "@drawable/icon",
  Theme = "@style/MainTheme", MainLauncher = true,
  ConfigurationChanges = ConfigChanges.ScreenSize | ConfigChanges.Orientation)]
public class MainActivity :
  global::Xamarin.Forms.Platform.Android.FormsAppCompatActivity
{
  protected override void OnCreate(Bundle bundle)
  {
    TabLayoutResource = Resource.Layout.Tabbar;
    ToolbarResource = Resource.Layout.Toolbar;

    base.OnCreate(bundle);

    global::Xamarin.Forms.Forms.Init(this, bundle);
    LoadApplication(new App());
  }
}
```

### 37.5.3　托管 Xamarin 的 iOS

托管 Xamarin.Forms 的 iOS 项目包含了 Main()方法，其中调用 UIApplication，传递 AppDelegate 字符串，类似于之前的 iOS。AppDelegate 类的实现是不同的。这个类现在派生自基类 Xamarin.Forms.Platform.iOS.FormsApplicationDelegate，FinishedLaunching 方法的实现初始化了 Xamarin.Forms，并加载应用程序(代码文件 BooksAppX/BooksAppX. iOS /AppDelegate.cs)：

```
[Register("AppDelegate")]
public partial class AppDelegate :
  global::Xamarin.Forms.Platform.iOS.FormsApplicationDelegate
{
  public override bool FinishedLaunching(UIApplication app,
    NSDictionary options)
  {
    global::Xamarin.Forms.Forms.Init();
    LoadApplication(new App());

    return base.FinishedLaunching(app, options);
  }
}
```

从这里开始，UI 的通用代码编写就可以开始了。

### 37.5.4　共享的项目

这个共享的项目包含了一个公共的 App 类，带有可重写的方法 OnStart、OnSleep 和 OnResume。这将映射到 Android、iOS 和 Windows 系统的生命周期。在构造函数中，创建了 MainPage(代码文件 BooksAppX /BooksAppX /App.xaml.cs)：

```
public partial class App : Application
{
  public App()
  {
    InitializeComponent();
    MainPage = new BooksAppX.MainPage();
  }

  protected override void OnStart()
  {
  }
  protected override void OnSleep()
  {
  }
  protected override void OnResume()
  {
  }
}
```

MainPage 的 XAML 内容类似于 Windows 应用程序，但它也有区别，其中使用了不同的 XAML 元素，如 ContentPage 和 Label。这些是用 Xamarin.Forms 定义的，在所有的 Xamarin.Forms 平台上都可以使用：

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
  xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
  xmlns:local="clr-namespace:BooksAppX"
  x:Class="BooksAppX.MainPage">
  <Label Text="Welcome to Xamarin.Forms!"
    VerticalOptions="Center"
    HorizontalOptions="Center" />
</ContentPage>
```

**注意：**

这里显示的 MainPage.xaml 包含创建新 Xamarin.Forms 时创建的 ContentPage。在可下载的源代码中，MainPage.xaml 包含一个 TabbedPage，参见 37.8 节“页面”。

## 37.6 使用公共库

要使用第 34 章创建的.NET 标准库 BooksLib 和 Framework，可以在 Xamarin.Forms 解决方案中添加来自文件夹 PatternsXamarinShared 的项目。这些项目需要从每个 Xamarin 项目中引用。

还需要添加 NuGet 包 Microsoft.Extensions.DependencyInjection 和 Microsoft.Extensions. Configuration。ILogger 接口由服务和视图模型使用。

Xamarin 还包括一个依赖注入容器，它可以从 DependencyService 类中访问。DependencyService.Register 注册一个服务，而 DependencyService.Get 从容器中返回服务。但是，因为 UWP 项目以及其他许多章节都在使用.NETCore DI 容器 Microsoft.Extensions.DependencyInjection，它也与 BooksAppX 一起使用。

可以实例化依赖注入容器，并可以在共享的 App 类中注册服务。对于具有特定于 UWP 实现的服务，现在需要完成特定于 Xamarin 的实现。需要 Xamarin 实现、特定于平台的服务是显示消息对话框的 IMessageService，和在页面之间导航的 INavigationService (代码文件 BooksAppX/ BooksAppX/App.xaml.cs)：

```
public partial class App : Application
{
  public App()
  {
    InitializeComponent();
    RegisterServices();

    MainPage = new BooksAppX.Views.MainPage();
  }
  //...

  private void RegisterServices()
  {
    var services = new ServiceCollection();
    services.AddSingleton<IBooksRepository, BooksSampleRepository>();
    services.AddSingleton<IItemsService<Book>, BooksService>();
    services.AddTransient<BooksViewModel>();
    services.AddTransient<BookDetailViewModel>();
    services.AddSingleton<IMessageService, XamarinMessageService>();
    services.AddSingleton<INavigationService, XamarinNavigationService>();
    services.AddSingleton<XamarinInitializeNavigationService>();
    services.AddLogging();
    AppServices = services.BuildServiceProvider();
  }

  public IServiceProvider AppServices { get; private set; }
}
```

IMessageService 和 INavigationService 的实现将在“导航”一节中讨论。

## 37.7 控件层次结构

第 33 章介绍了 Windows 应用程序控件的主要层次。Xamarin.Forms 有类似的层次结构，但它有很大的区别，需要在这里讨论。此外，这些控件的名称也不同。

下面介绍 Xamarin.Forms 层次结构中最重要的类。

- BindableObject：这是 Xamarin.Forms 控件的基类。这个类实现了 INotifyPropertyChanged 接口。在 UWP 中没有此类，但是 BindableBase 类有一个定制的实现。BindableObject 还实现了接口 IDynamicResourceHandler。该接口由平台渲染器使用。
- Element：Element 类作为 Forms 层次结构中所有元素的基类。这个类定义了遍历树的方法和属性，并实现了平台渲染器的功能。
- Cell：Cell 类派生自 Element。可以将 Cell 添加到 ListView 和 TableView 类中。与可以使用 UWP 进行的操作不同，不能将任何 XAML 元素添加到 ListView 中。在 Xamarin.Forms 中，只能添加从 Cell 中派生的类。

- VisualElement：VisualElement 派生自 Element，是所有有可视化外观的元素的基类。在 UWP 中，Height 和 Width 属性可以定义所需的尺寸，并根据容器的大小来安排。而在 Xamarin.Forms 中，Height 和 Width 属性都是只读的，像 UWP 中的 ActualHeight和 ActualWidth 一样使用。Xamarin.Forms 中的 HeightRequest 和 WidthRequest 是 UWP 中 Height 和 Width 属性的变体。VisualElement 类也有一个 Style 属性，可以定义缩放、旋转和不透明度，可以通过 IsEnabled 设置要启用的元素，通过 IsVisible 设置元素是否显示。
- Page：Page 类派生自 VisualElement，是所有页面的基类。这个类定义了标题(title 属性)、工具栏(ToolbarItems)、图标(Icon)。使用方法 DisplayAlert，可以显示一个带有 accept 和 cancel 按钮的对话框。
- View：View 类派生自 VisualElement，是几乎所有控件的基类。这个类定义了指定在布局中如何使用元素的水平和垂直选项、定义了视图页边距的 Margin 属性，以及手势识别器。
- ItemsView：ItemsView 派生自 View，可以与 UWP 的 ItemsControl 类进行比较。ItemsView 是 ListView 的基类，定义了 ItemsSource 和 ItemTemplate 属性。
- InputView：InputView 派生自 View，是允许键盘输入的基类。这个类有一个 Keyboard 属性。
- Layout：Layout 类派生自 View，允许派生自 Layout 的类定位它的子元素。在 UWP 中，这是 Panel 类。

接下来讨论这些控件和类别。

## 37.8 页面

要创建用户界面，应从 MainPage 开始。Windows 应用程序为主页使用 NavigationView。此控件不可用于 Xamarin.Forms。记住，Xamarin.Forms 只提供了可映射到所有平台的控件。

可用的一个页面类型是 TabbedPage。在 UWP 中，TabbedPage 就是 Pivot 控件，在 iOS 中它是 UiTabBarController。TabbedPage 包含 AboutPage，它本身就是 ContentPage 和 NavigationPage。NavigationPage 相当于 Windows 应用程序的 Frame。NavigationPage 的内容在导航时替换(代码文件 BooksAppX/BooksAppX/Views/ MainPage.xaml)：

```
<?xml version="1.0" encoding="utf-8" ?>
<TabbedPage xmlns="http://xamarin.com/schemas/2014/forms"
  xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
  x:Class="BooksAppX.Views.MainPage">
  <TabbedPage.Children>
    <NavigationPage Title="Books">
      <NavigationPage.Icon>
        <OnPlatform x:TypeArguments="FileImageSource">
          <On Platform="iOS" Value="tab_feed.png"/>
        </OnPlatform>
      </NavigationPage.Icon>
      <x:Arguments>
        <views:BooksPage x:Name="booksPage" />
      </x:Arguments>
    </NavigationPage>
    <views:AboutPage Title="About" />
  </TabbedPage.Children>
</TabbedPage>
```

该应用程序的其他页面——BooksPage 和 BooksDetailPage—使用简单的 ContentPage 页面。ContentPage 可以包含一个内容。

可用于 Xamarin.Forms 的页面如表 37-3 所示。

表 37-3

| Xamarin.Forms 的页面类 | UWP 类 | 说　明 |
|---|---|---|
| TabbedPage | Pivot | TabbedPage 提供多个选项卡在页面之间导航。这是一个常用的用户界面 |
| NavigationPage | Frame | NavigationPage 提供了像 UWP 中 Frame 类的导航功能。Xamarin.Forms 的 Frame 类可以与 UWP 的 Border 类相比较 |

(续表)

| Xamarin.Forms 的页面类 | UWP 类 | 说　明 |
|---|---|---|
| ContentPage | 无 | ContentPage 派生自 TemplatedPage，而 TemplatedPage 派生自 Page。使用模板页面，可以分配 ControlTemplate 属性来定义内容。ContentPage 只能有一个内容 |
| MasterDetailPage | 无 | MasterDetailPage 显示了一个列表和详细信息。不同的页面可以用于主要和细节部分 |
| CarouselPage | FlipView | CarouselPage 派生自 MultiPage。这个页面允许用户浏览多个页面。这种功能在 UWP 中很常见，但 Android 和 iOS 却不常见 |

## 37.9　导航

该应用程序的根页面包括 TabbedPage，其中包括 NavigationPage。NavigationPage 定义的属性 Navigation 返回初始化 INavigation 接口的对象。这样，就可以使用 PushAsync 方法导航到另一个页面，并使用 PopAsync 方法返回。

使用.NET 标准库 Framework，用这些方法定义 INavigationService：

```
public interface INavigationService
{
  Task NavigateToAsync(string page);
  Task GoBackAsync();
  string CurrentPage { get; }
}
```

现在，需要 Xamarin.Forms 的具体实现。这个初始化给导航界面使用了 INavigation 接口(代码文件 BooksAppx/Services/XamarinNavigationService.cs)：

```
public class XamarinNavigationService : INavigationService
{
  private Dictionary<string, Func<Page>> _pages =
    new Dictionary<string, Func<Page>>
  {
    [PageNames.BookDetailPage] = () => new BookDetailPage()
  };

  private INavigation _navigation;
  private INavigation Navigation
  {
    get => _navigation ?? (_navigation = _initializeNavigation.Navigation);
  }

  public string CurrentPage => throw new NotImplementedException();

  private readonly XamarinInitializeNavigationService _initializeNavigation;

  public XamarinNavigationService(
    XamarinInitializeNavigationService initializeNavigation)
  {
    _initializeNavigation = initializeNavigation;
  }

  public Task GoBackAsync() => Navigation.PopAsync();

  public Task NavigateToAsync(string pagename) =>
    Navigation.PushAsync(_pages[pagename]());
}
```

现在需要为初始化提供 XamarinInitializeNavigationService。这个类只定义了 XamarinNavigationService 类使用的 SetNavigation 方法和 Navigation 属性(代码文件 BooksAppX/Services/XamarinInitializeNavigationService.cs)：

```
public class XamarinInitializeNavigationService
{
  public void SetNavigation(INavigation navigation) =>
    _navigation = navigation;
```

```
  private INavigation _navigation;
  public INavigation Navigation => _navigation ??
    throw new ArgumentException("navigation not initialized");
}
```

现在，需要从 NavigationPage 可用的页面中设置 Navigation 属性——MainPage 的代码隐藏文件(代码文件 BooksAppX/Views/MainPage.xaml.cs)。

```
[XamlCompilation(XamlCompilationOptions.Compile)]
public partial class MainPage : TabbedPage
{
  public MainPage ()
  {
    InitializeComponent();

    (Application.Current as App).AppServices
      .GetService<XamarinInitializeNavigationService>()
      .SetNavigation(navigationPage.Navigation);
  }
}
```

## 37.10 布局

About 页面使用 Grid 和 StackLayout 控件来安排其子元素。Grid 控件允许定义行和列——就像使用 UWP 中的 Grid 控件一样——使用自动设置、星形设置和固定大小设置(代码文件 BooksAppX/Views/AboutPage.xaml)：

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
  xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
  x:Class="BooksAppX.Views.AboutPage">
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto" />
      <RowDefinition Height="*" />
    </Grid.RowDefinitions>
    <StackLayout BackgroundColor="{StaticResource Accent}"
      VerticalOptions="FillAndExpand" HorizontalOptions="Fill">
      <StackLayout Orientation="Horizontal" HorizontalOptions="Center"
        VerticalOptions="Center">
        <ContentView Padding="0,40,0,40" VerticalOptions="FillAndExpand">
          <Image Source="xamarin_logo.png" VerticalOptions="Center"
            HeightRequest="64">
            <Image.Source>
              <OnPlatform x:TypeArguments="ImageSource">
                <On Platform="UWP" Value="Assets/xamarin_logo.png" />
              </OnPlatform>
            </Image.Source>
          </Image>
        </ContentView>
      </StackLayout>
    </StackLayout>
    <ScrollView Grid.Row="1">
      <StackLayout Orientation="Vertical" Padding="16,40,16,40" Spacing="10">
        <Label FontSize="22">
          <Label.FormattedText>
            <FormattedString>
              <FormattedString.Spans>
                <Span Text="AppName" FontAttributes="Bold" FontSize="22" />
                <Span Text=" " />
                <Span Text="1.0"
                  ForegroundColor="{StaticResource LightTextColor}" />
              </FormattedString.Spans>
            </FormattedString>
          </Label.FormattedText>
        </Label>
        <Label>
          <Label.FormattedText>
            <FormattedString>
              <FormattedString.Spans>
                <Span Text="This is a sample app for " />
                <Span Text="Professional C# 7 and .NET Core 2.0"
                  FontAttributes="Bold" />
```

```
                <Span Text="." />
              </FormattedString.Spans>
            </FormattedString>
          </Label.FormattedText>
        </Label>
        <Label>
          <Label.FormattedText>
            <FormattedString>
              <FormattedString.Spans>
                <Span Text="It shares code with" />
                <Span Text=" " />
                <Span Text="iOS, Android, and Windows" FontAttributes="Bold" />
                <Span Text="." />
              </FormattedString.Spans>
            </FormattedString>
          </Label.FormattedText>
        </Label>
        <Button Margin="0,10,0,0" Text="Learn more" Command="{Binding
          ViewModel.OpenWebCommand, Mode=OneWay}"
          BackgroundColor="{StaticResource Primary}" TextColor="White" />
      </StackLayout>
    </ScrollView>
  </Grid>
</ContentPage>
```

表 37-4 列出了 Xamarin.Forms 的布局控件，以及它们与 Windows 面板控件的比较。

表 37-4

| Xamarin.Forms 控件 | UMP 控 件 | 说 明 |
|---|---|---|
| Grid | Grid | 使用星型、自动和固定大小来组织行和列中的内容 |
| StackLayout | StackPanel | 把子控件一个接一个地水平或垂直排列起来 |
| AbsoluteLayout | Canvas | 使用附加属性 AbsoluteLayout.LayoutBounds 和 AbsoluteLayout.LayoutFlags，通过绝对坐标定位子元素 |
| RelativeLayout | RelativePanel | 相对于其他控件安排控件 |

About 页面还有其他一些有趣的方面。使用 OnPlatform 元素，可以根据平台定义不同的 XAML 元素和属性。使用 Image 元素，将 Source 属性设置为一个 PNG 文件。该文件对 Android 和 iOS 是有效的，但是对于 Windows，该文件存储在 Assets 文件夹中。只有 UWP 中 Source 属性的设置是不同的，它由 OnPlatform 元素定义：

```
<Image Source="xamarin_logo.png" VerticalOptions="Center"
  HeightRequest="64">
  <Image.Source>
    <OnPlatform x:TypeArguments="ImageSource">
      <On Platform="UWP" Value="Assets/xamarin_logo.png" />
    </OnPlatform>
  </Image.Source>
</Image>
```

另一个有趣的部分是 Label 元素。与 UWP 中的 TextBlock 一样，Label 不仅可以包含简单的文本，还可以包含用不同字体和颜色格式化的文本。参见第 36 章，以获得更多关于 TextBlock 的文本格式信息。使用带有 Label 的不同文本格式，需要将 FormattedText 属性设置为 FormattedString：

```
<Label>
  <Label.FormattedText>
    <FormattedString>
      <FormattedString.Spans>
        <Span Text="This is a sample app for " />
        <Span Text="Professional C# 7 and .NET Core 2.0"
          FontAttributes="Bold" />
        <Span Text="." />
      </FormattedString.Spans>
    </FormattedString>
  </Label.FormattedText>
</Label>
```

## 37.11　视图

下面讨论一些视图类型，并将它们与已知的控件进行比较，如表 37-5 所示。

表 37-5

| Xamarin.Forms 控件 | UWP 控件 | 说　明 |
|---|---|---|
| Button | Button | 按钮会对触摸事件做出反应，就像 UWP 中的按钮一样。但是，按钮不能包含任何内容——只能包含文本 |
| Label | TextBlock | 标签不仅可以包含文本，像 TextBlock 一样，还可以包含格式化的文本，如前一节所述 |
| Editor Entry | TextBox | 与 UWP 不同，Xamarin.Forms 对多行编辑和单行编辑有不同的控制。要只编辑一行，可以使用 Entry 类。Editor 支持多行编辑 |
| SearchBar | AutoSuggestBox | UWP 使用 AutoSuggestBox 渲染 SearchBar |
| Stepper | 无 | UWP 中没有与 Stepper 等效的控件。在 Xamarin.Forms 中，这是通过用户控件和两个按钮实现的 |
| ListView | ListView | Xamarin.Forms 定义了一个具有与 UWP 中的 ListView 类似功能的 ListView，来使用 ItemTemplate 显示项列表 |
| ProgressBar | ProgressBar | Xamarin.Forms 定义 ProgressBar 来显示进度信息，类似于 UWP |
| Switch | ToggleSwitch | Switch 类类似于 UWP 中的 ToggleSwitch |

**注意：**

Xamarin.Forms 和 UWP 有许多相似之处，但类名、属性名和方法名有许多不同之处。因为有.NET 标准，所以现在，XAML 标准正在使用中：参见 https://github.com/ Microsoft/ xaml-standard。我们需要检查问题，以查看实际进展。在写这篇文章的时候，对 Xamarin.Forms 来说，可以使用一个预览包，该包允许使用 XAML 标准和 XAML 文件。这样，XAML 代码就从 XAML 标准转换到 Xamarin.Forms 控件。Xamarin 控件本身不会随 XAML 标准而改变，但 XAML 语法相同。

## 37.12　数据绑定

Xamarin.Forms 的数据绑定是使用标记扩展完成的。x: Bind 标记扩展不可用，但是可以使用 Binding 标记扩展，它也可以与 UWP 一起使用。请注意，默认模式是不同的，可以使用的模式也是不同的。Xamarin.Forms 不支持 OneTime，但 OneWay、TwoWay 和 OneWayToSource 都不能用于 UWP。此模式将来自源的数据绑定到目标(UI 元素)，但反之无效。下面的代码片段绑定了 Entry 控件的 Text 和 IsEnabled 属性(代码文件 BooksAppX/Views/ BookDetailPage.xaml)：

```
<ContentPage.Content>
  <StackLayout Spacing="20" Padding="15">
    <Label Text="Id:" FontSize="Medium" />
    <Entry IsEnabled="False"
      Text="{Binding ViewModel.EditItem.BookId, Mode=OneWay}" />
    <Label Text="Title:" FontSize="Medium" />
    <Entry IsEnabled="{Binding ViewModel.IsEditMode, Mode=OneWay}"
      Text="{Binding ViewModel.EditItem.Title, Mode=TwoWay}"
      FontSize="Small" />
    <Label Text="Publisher:" FontSize="Medium" />
    <Entry IsEnabled="{Binding ViewModel.IsEditMode, Mode=OneWay}"
      Text="{Binding ViewModel.EditItem.Publisher, Mode=TwoWay}"
      FontSize="Small" />
  </StackLayout>
</ContentPage.Content>
```

该属性没有指定 DataContext 来激活绑定，而是具有名称 BindingContext(代码文件 BooksAppX /视图/BookDetailPage.xaml.cs)：

```
[XamlCompilation(XamlCompilationOptions.Compile)]
public partial class BookDetailPage : ContentPage
{
  public BookDetailPage()
  {
    InitializeComponent();
    BindingContext = this;
  }

  public BookDetailViewModel ViewModel { get; } =
    (Application.Current as App).AppServices.GetService<BookDetailViewModel>();
}
```

# 37.13 命令

与 UWP 一样，Xamarin.Forms 中的命令实现了 ICommand 接口。Xamarin.Forms 已经通过 Command 类包含了一个实现。Command 类的实现类似于 Framework 库中实现的 RelayCommand。Command 用于 AboutViewModel(代码文件 BooksAppX/ViewModels /AboutViewModel.cs)：

```
public class AboutViewModel
{
  public AboutViewModel()
  {
    OpenWebCommand = new Command(() =>
      Device.OpenUri(new Uri("https://csharp.christiannagel.com")));
  }

  public ICommand OpenWebCommand { get; }
}
```

Button 类的 Command 属性是通过数据绑定来分配的(代码文件 BooksAppX/Views/ AboutPage.xaml)：

```
<Button Margin="0,10,0,0" Text="Learn more"
  Command="{Binding ViewModel.OpenWebCommand, Mode=OneWay}"
  BackgroundColor="{StaticResource Primary}" TextColor="White" />
```

Xamarin.Forms 还支持 ToolBarItem 控件，该控件可分配给 ContentPage 的 ToolBarItems 属性。这些命令绑定到在 BooksLib 库的视图模型中定义的命令(代码文件 BooksAppX/Views/BooksPage.xaml)：

```
<ContentPage.ToolbarItems>
  <ToolbarItem Text="Refresh"
    Command="{Binding ViewModel.RefreshCommand, Mode=OneWay}" />
  <ToolbarItem Text="Add Book"
    Command="{Binding ViewModel.AddCommand, Mode=OneWay}" />
</ContentPage.ToolbarItems>
```

# 37.14 ListView 和 ViewCell

ListView 控件用于显示项列表。在示例应用程序中，ItemsSource 属性绑定到视图模型的 Items 属性上。为了进行显示，将 DataTemplate 分配给 ItemTemplate 属性。这看起来非常类似于 UWP，唯一的区别是 ListView 中项的 DataTemplate 只能包含派生自 Cell 的类。这里使用了一个 ViewCell，以允许使用其他元素，如 StackLayout 和 Label 控件(代码文件 BooksAppX/Views/ BooksPage.xaml)：

```
<ListView x:Name="BooksListView"
  ItemsSource="{Binding ViewModel.Items}"
  VerticalOptions="FillAndExpand"
  HasUnevenRows="true"
  RefreshCommand="{Binding ViewModel.RefreshCommand}"
  IsPullToRefreshEnabled="true"
  IsRefreshing="{Binding IsBusy, Mode=OneWay}"
  CachingStrategy="RecycleElement"
  SelectedItem="{Binding ViewModel.SelectedItem, Mode=TwoWay}">
  <ListView.ItemTemplate>
```

```
        <DataTemplate>
          <ViewCell>
            <StackLayout Padding="10">
              <Label Text="{Binding Title}"
                LineBreakMode="NoWrap"
                Style="{DynamicResource ListItemTextStyle}"
                FontSize="16" />
            </StackLayout>
          </ViewCell>
        </DataTemplate>
      </ListView.ItemTemplate>
    </ListView>
```

其他派生自 Cell 的类已经包含了预定义的功能，所以不需要添加 Entry Label 控件。表 37-6 列出了派生自 Cell 的类。

表 37-6

| Cell | 说　明 |
|---|---|
| EntryCell | 允许编辑一行的单元格，类似于 Entry 控件 |
| SwitchCell | 带有标签和开关的选项 |
| TextCell | 主要和次要文本 |
| ImageCell | 带有图像的单元格 |
| ViewCell | 开发人员定义的视图，如 BooksPage 示例所示 |

**注意：**

可以在 Windows、iOS 和 Android 设备的屏幕截图中看到正在运行的应用程序，这些设备都在本书的 GitHub 站点上，网址是：

https://github.com/ ProfessionalCSharp/ProfessionalCSharp7/blob/master/Xamarin/Readme.md。

# 37.15　小结

本章介绍了如何使用 Xamarin 开发移动应用程序，学习了 Android 和 iPhone 的不同方面，以及如何使用 Xamarin.Android 和 Xamarin.iOS 开发应用程序。

本章的重点是 Xamarin.Forms，它如何提供与 UWP 相同的功能，以及它们的区别。虽然元素名称和属性与 UWP 不同，但功能上的区别并不大——可以只实现一次 XAML 代码，就获得用于 Windows、iOS 和 Android 的应用程序。

在视图模型和服务中实现的应用程序的完整功能是在本地的 UWP 应用程序和 Xamarin.Forms 之间共享的。对于 Xamarin.Forms，只需要为用户界面定义顶部的小层。